An introduction to ROS

matteo.luperto@unimi.it

ROS: the Robot Operating System

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. [wiki.ros.org]

Robot software architecture

Low level functionalities as real-time motor controllers, sensors drivers, battery management

+

Core functionalities as mapping, localization, navigation, people detection

+

Reasoning mechanism for path planning, task allocation, self management

Robot software architecture

The development of (even a single) robots (functionality) requires both low-level hardware related and high-level AI-based mechanism

Modularity and scalability are consequently core features in a robot software architecture

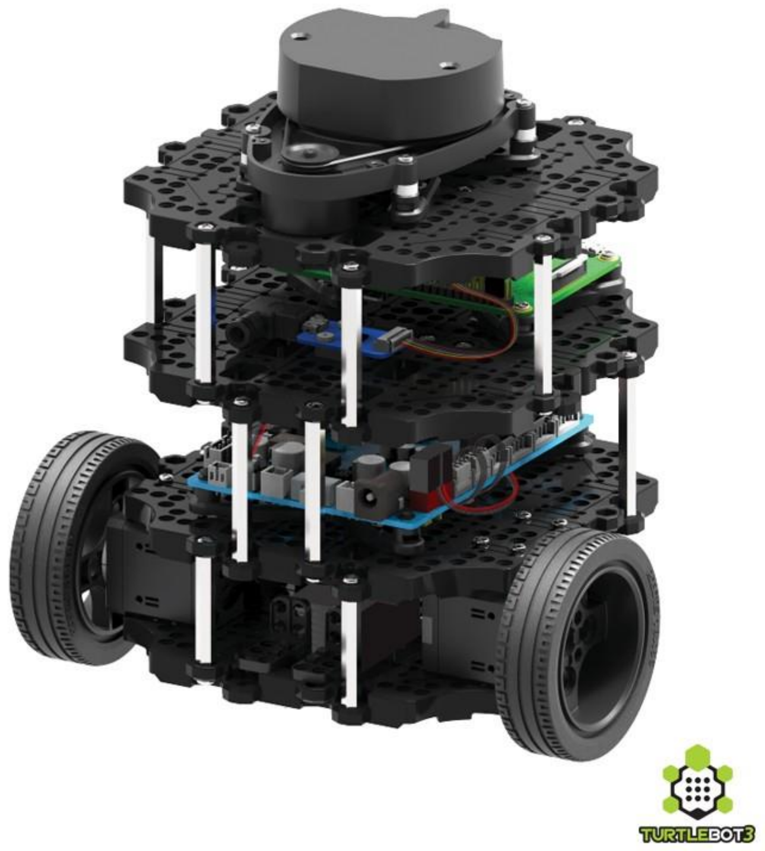
ROS provide this

ROS has established itself as the de-facto standard for robot development

ROS

More at:
robots.ros.org

Our ROS robots



Sensors with ROS [wiki.ros.org]

What is ROS?

Is a Meta-Operating System

- Scheduling – loading – monitoring, and error handling
- virtualization layer between applications and distributing computing resources
- runs on top of (multiple) operating system(s)

- is a framework
- not a real-time framework but embed real-time code
- enforce supports a modular software architecture

ROS SW architecture

- distributed framework of processes (*Nodes*)
- enables executables to be individually designed and loosely coupled at runtime.
- processes can be easily shared and distributed.
- supports a federated system of code *Repositories* that enable collaboration to be distributed as well.

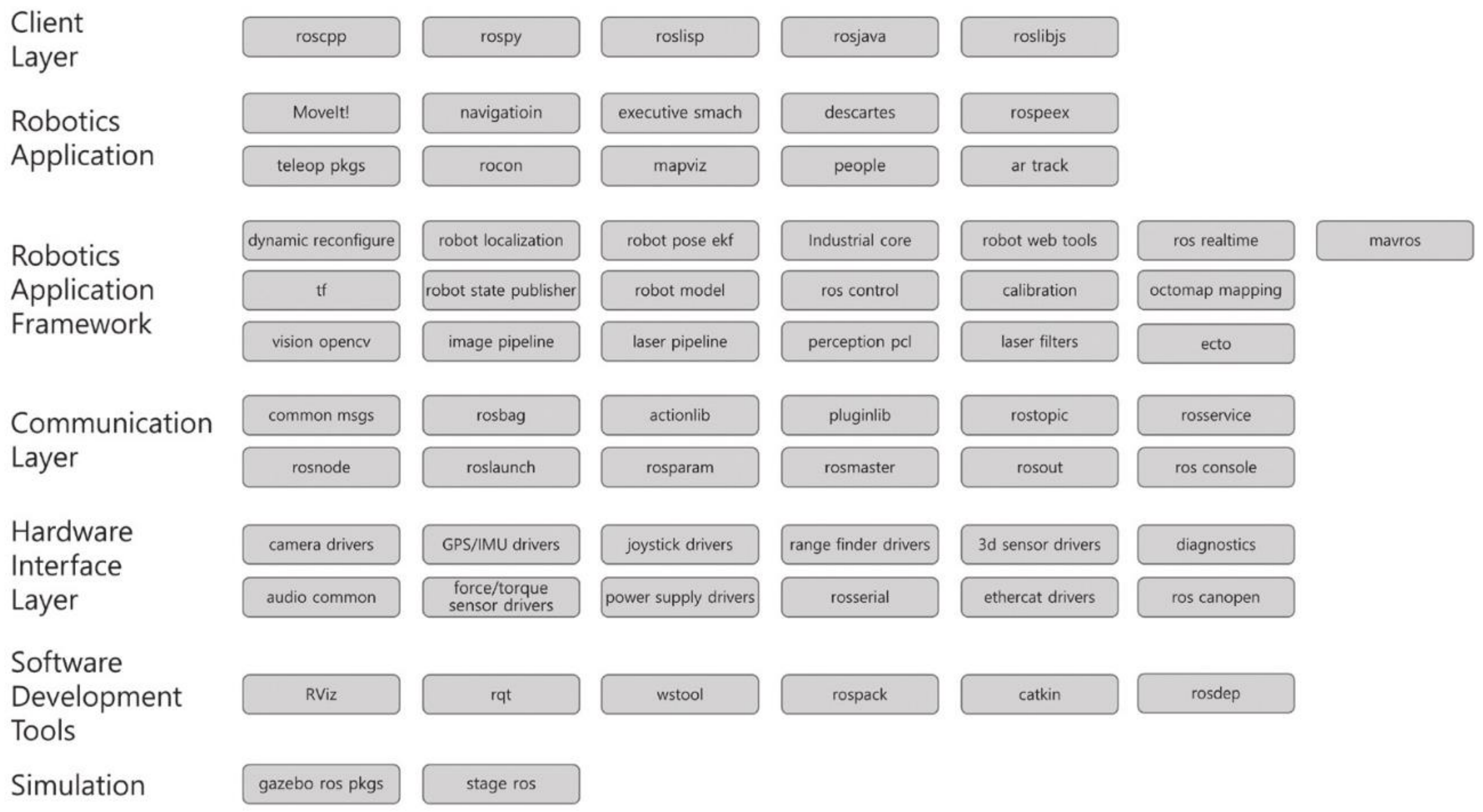
This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

More ROS features

- thin: ROS is designed to be as thin as possible
- easy to integrate with other frameworks and libraries
- language independence
  core languages are Python and C++ but you can use what you want
- easy testing: built in unit/integration test framework and debug tool
- scaling: ROS tools can be distributed across different machines and is appropriate for large development process

The core idea behind all of this is: code reuse + modularity

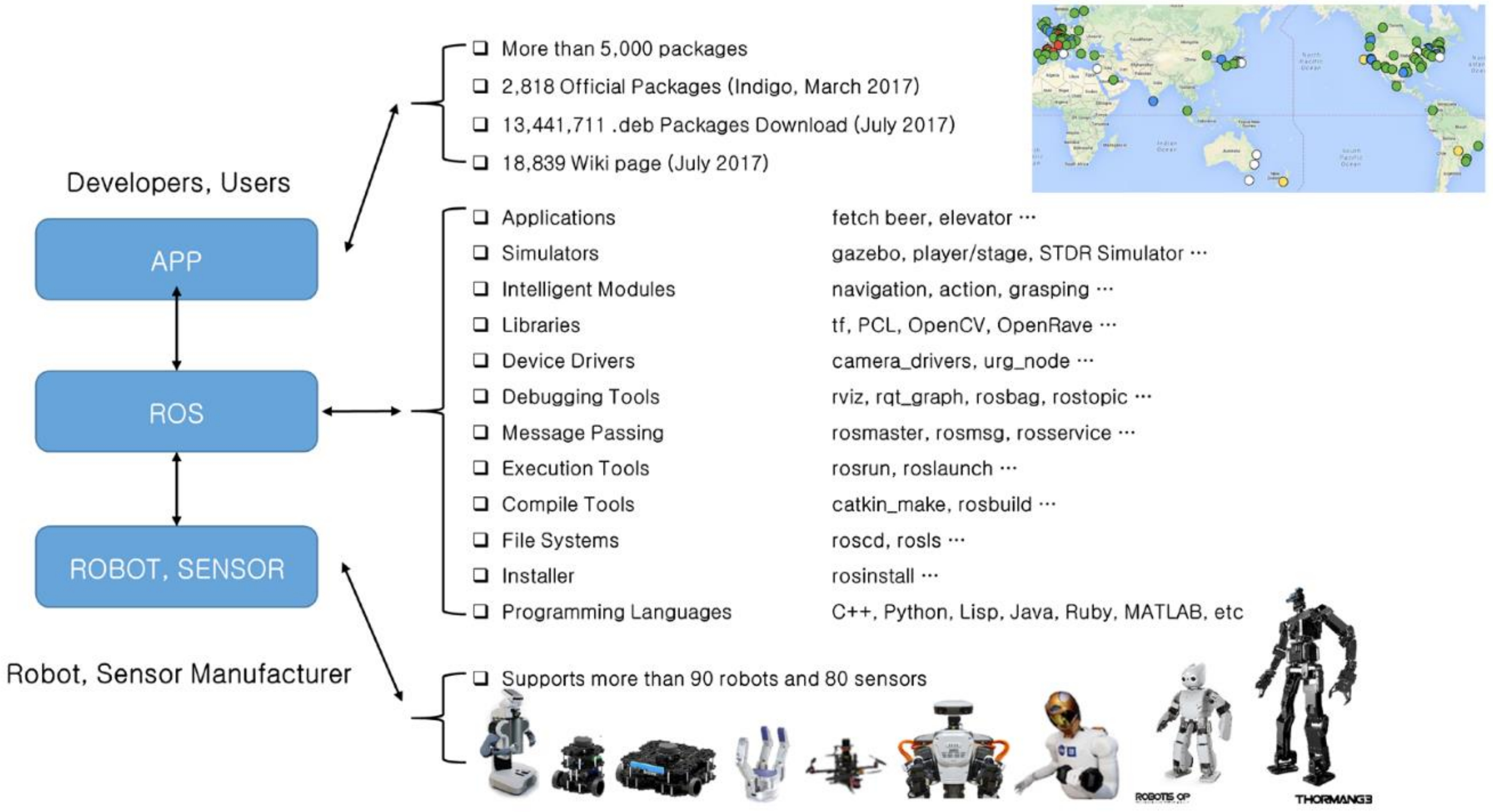
What ROS provides



- core and advanced robot functionalities (mapping, localization, navigation, obstacle avoidance)
- drivers and integration with sensors
- integration with multiple robot architectures
  UAV – manipulators –wheeled robots
- integration with libraries (OpenPose, OpenCV, deep learning fw)
- simulation tools

All free and ready to use
Support from the community

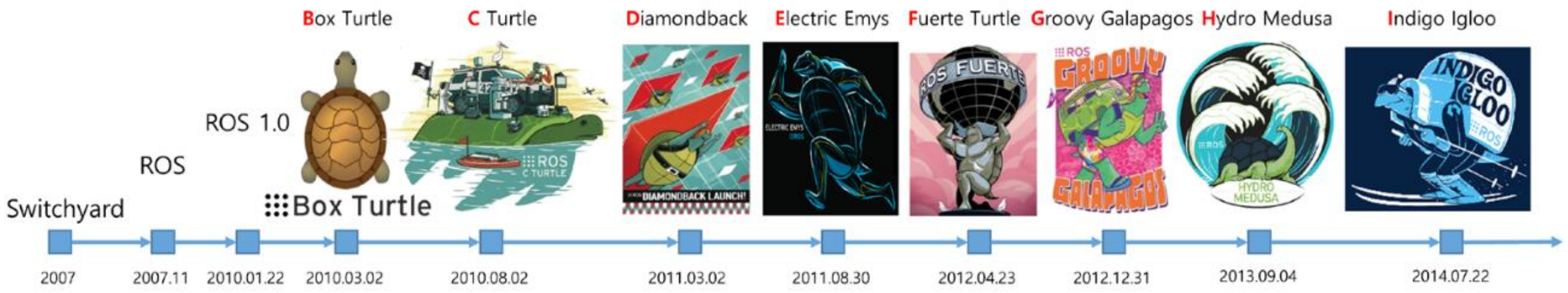
ROS-community

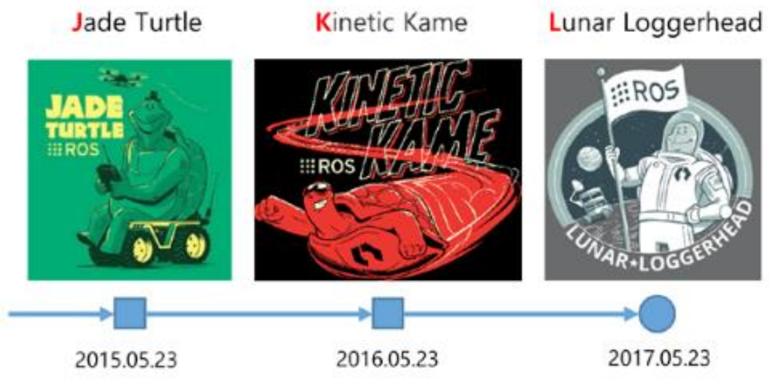


- ❑ More than 5,000 packages
- ❑ 2,818 Official Packages (Indigo, March 2017)
- ❑ 13,441,711 .deb Packages Download (July 2017)
- ❑ 18,839 Wiki page (July 2017)

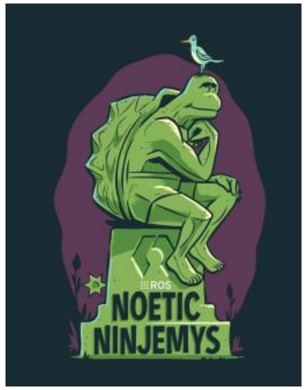
❑ Applications	fetch beer, elevator ⋯
❑ Simulators	gazebo, player/stage, STDR Simulator ⋯
❑ Intelligent Modules	navigation, action, grasping ⋯
❑ Libraries	tf, PCL, OpenCV, OpenRave ⋯
❑ Device Drivers	camera_drivers, urg_node ⋯
❑ Debugging Tools	rviz, rqt_graph, rosbag, rostopic ⋯
❑ Message Passing	rosmaster, rosmsg, rosservice ⋯
❑ Execution Tools	rosrun, roslaunch ⋯
❑ Compile Tools	catkin_make, rosbuild ⋯
❑ File Systems	roscd, rosls ⋯
❑ Installer	rosinstall ⋯
❑ Programming Languages	C++, Python, Lisp, Java, Ruby, MATLAB, etc

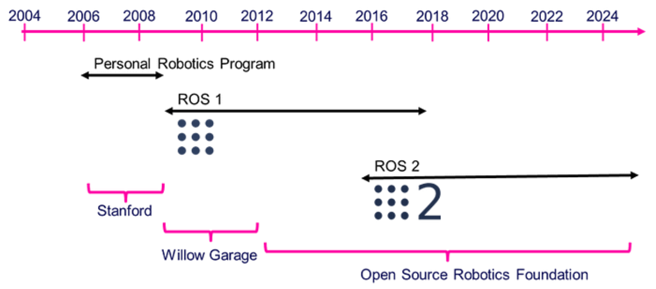
- ❑ Supports more than 90 robots and 80 sensors

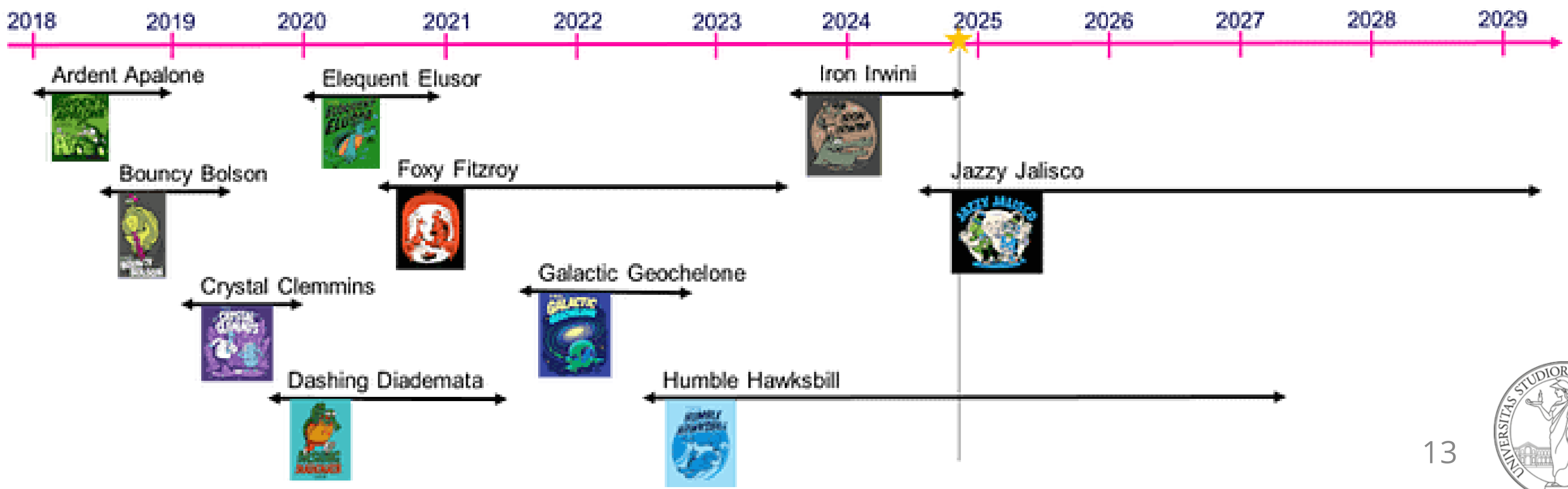




- more than 10y of ROS now
- last version (ROS1): ROS Noetic (2020)
- **Now: ROS2**







Core aspects of ROS

ROS aspects

- nodes
- topics
- messages

Building blocks of ROS

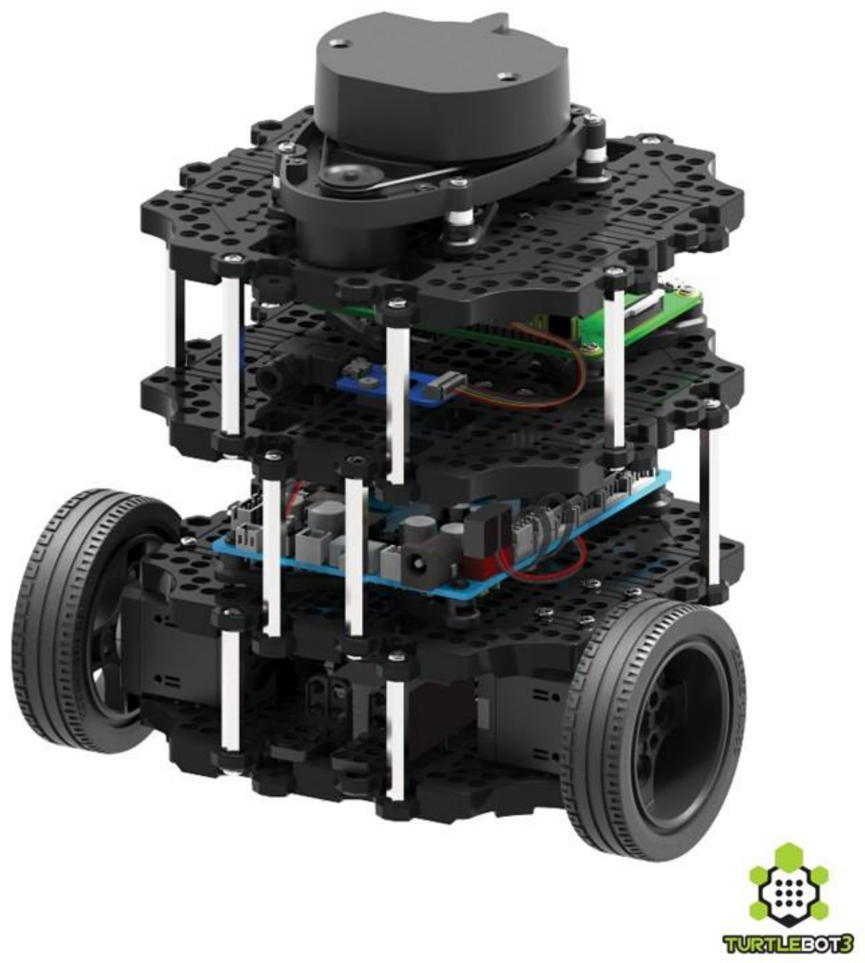
- services
- actions
- transforms

Communication / SW architecture

- debugging Tools
- simulations
- bags

Developers tools

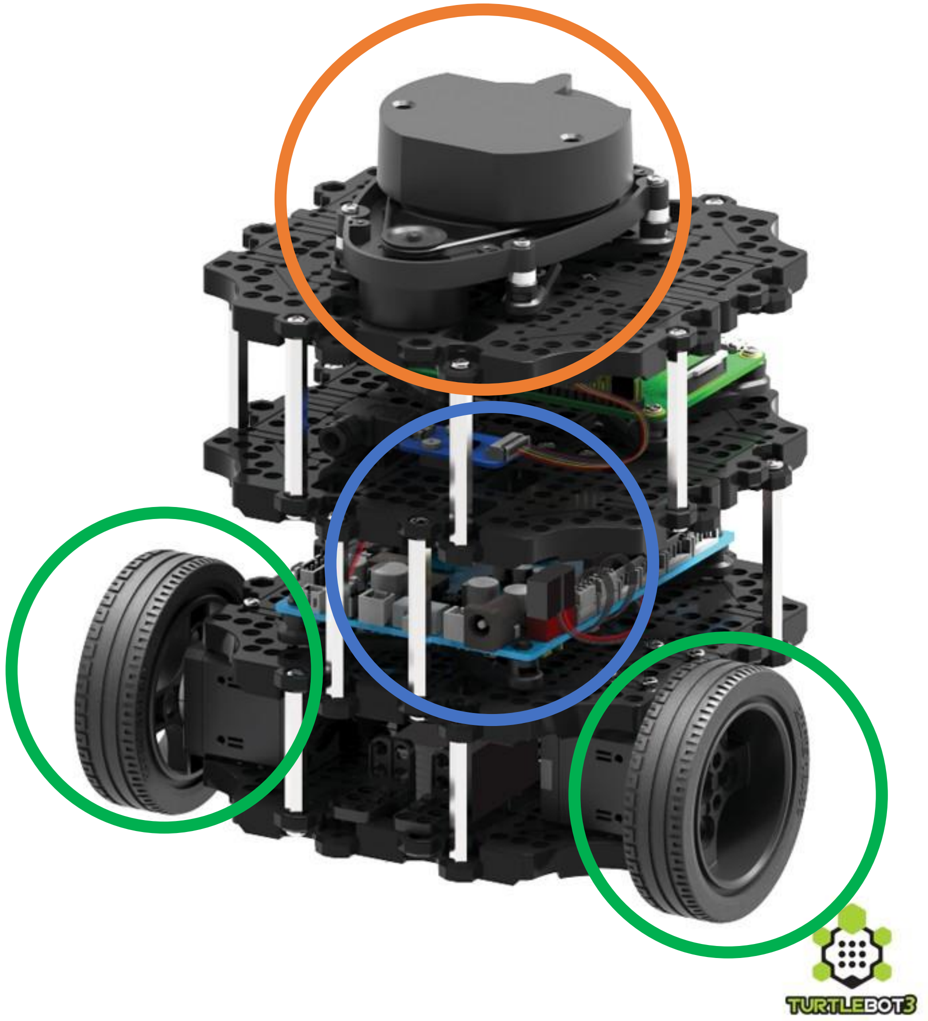
ROS *nodes*



A *node* is a process that performs computation:

- nodes are combined together into a graph and communicate with one another using streaming topics, services, and parameters,
- are meant to operate at a fine-grained scale,
- a robot control system will usually comprise many nodes.

ROS *nodes*



For example, one node controls a laser range-finder, one Node controls the robot's wheels motors, one node performs mapping, one localization, one node performs path planning, one node gives velocity commands to the wheels, one node provides a graphical view of the system, and so on.

ROS *nodes*

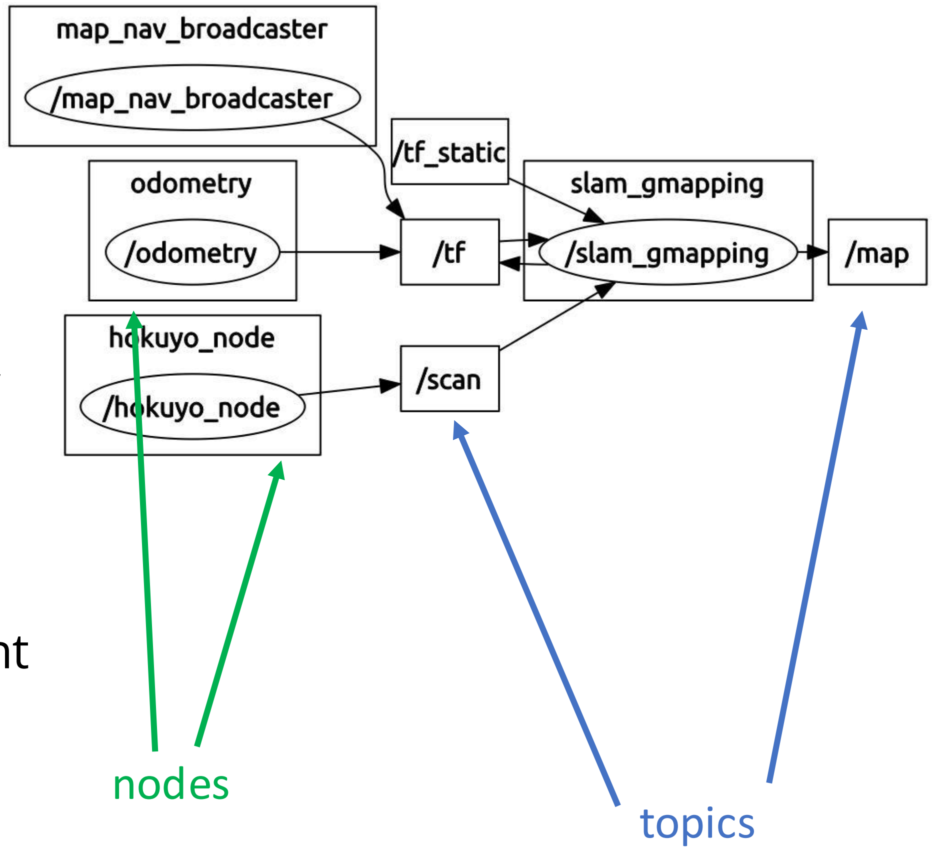
The use of nodes in ROS provides several benefits to the overall system.

- *fault tolerance* as crashes are isolated to individual nodes.
- *code complexity* is reduced in comparison to monolithic systems. Implementation details are also well hidden - nodes expose a minimal API –
- alternate implementations, even in other programming languages, can easily be substituted.

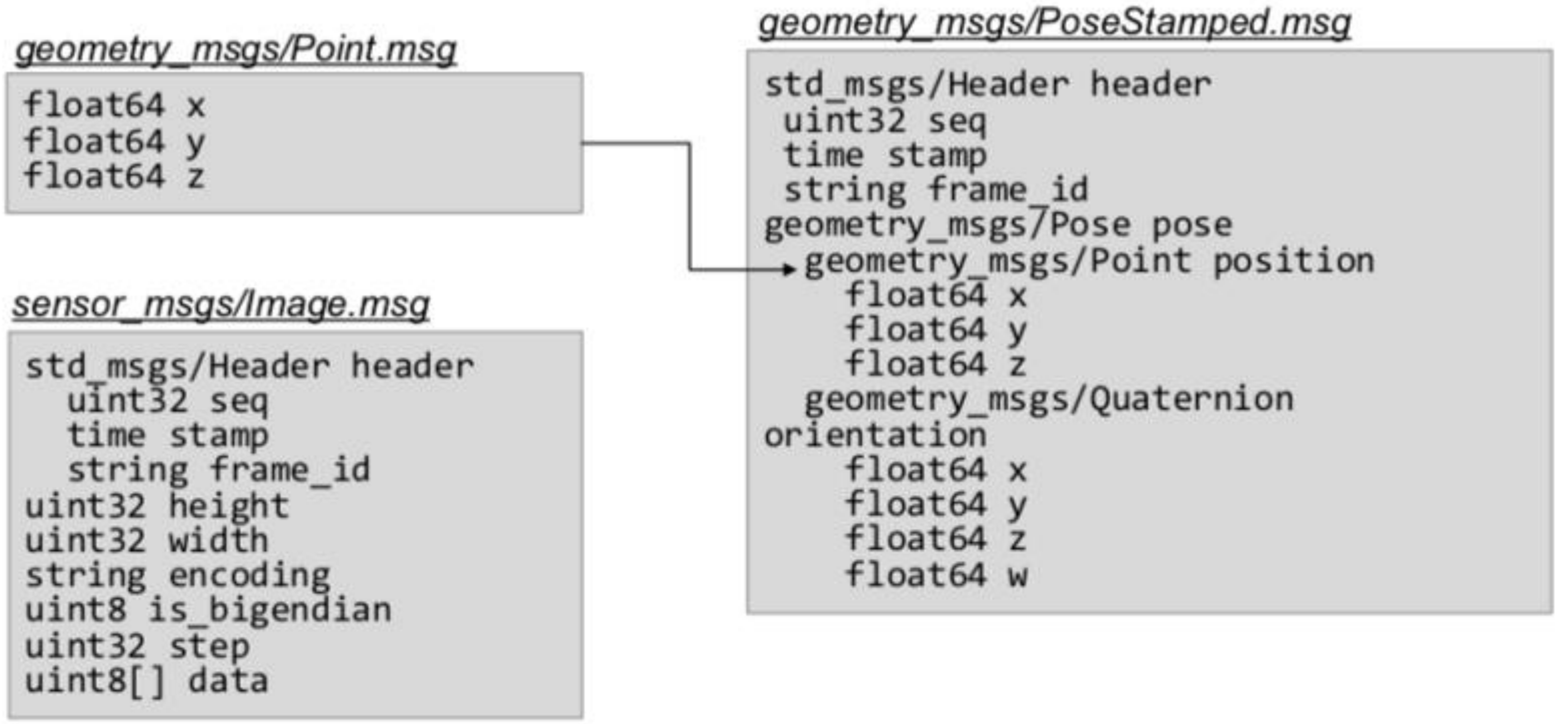
Nodes and *topics*

Topics are named buses over
which nodes exchange messages.

- topics have **anonymous publish/subscribe semantics**, which decouples the production of information from its consumption.
- nodes are not aware of who they are communicating with.
- nodes that are interested in data *subscribe* to the relevant topic; nodes that generate data *publish* to the relevant topic.
- there can be multiple publishers and subscribers to a topic.



ROS *topics* and *messages*


geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

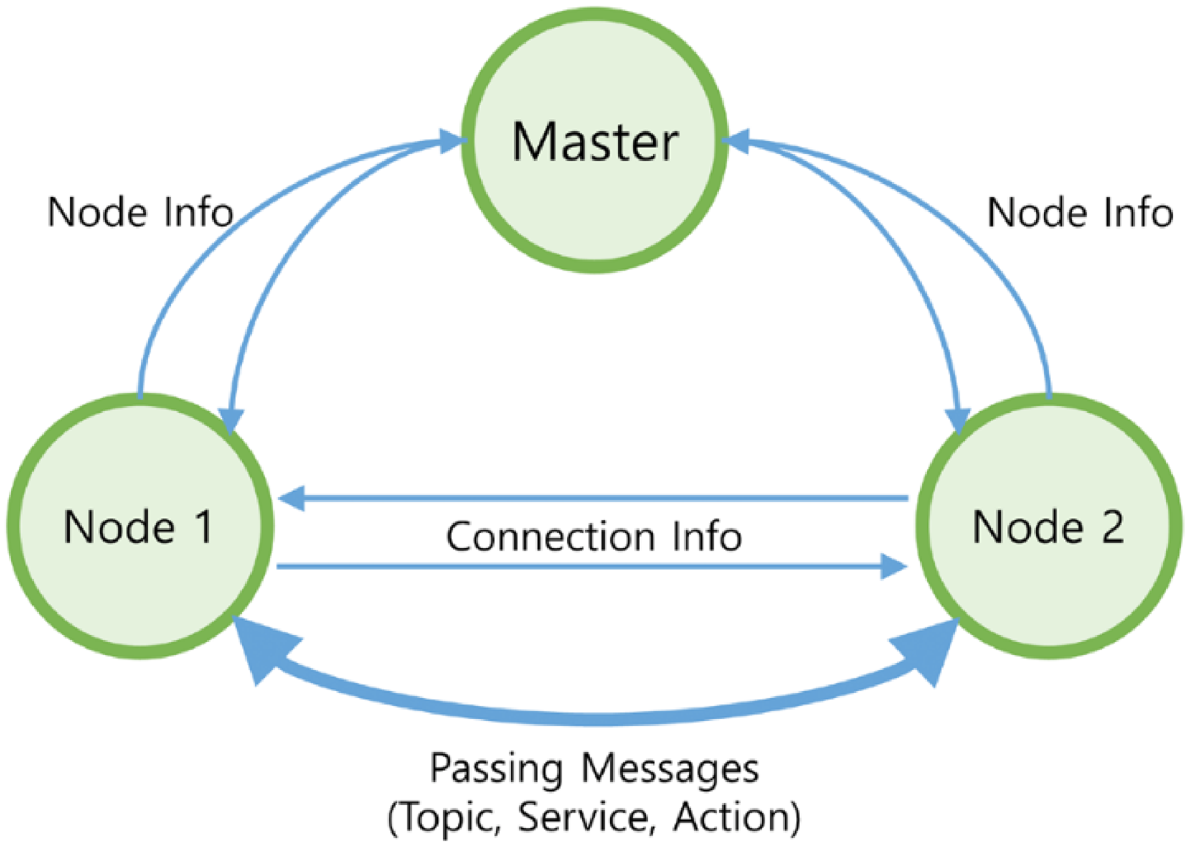
geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion
orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

- each topic is strongly typed by the ROS message type used to publish to it
- nodes can only receive messages with a matching type.
- type consistency is not enforced among the publishers, but subscribers will not establish message transport unless the types match.
- all ROS clients check to make sure that an MD5 computed from the message format match.

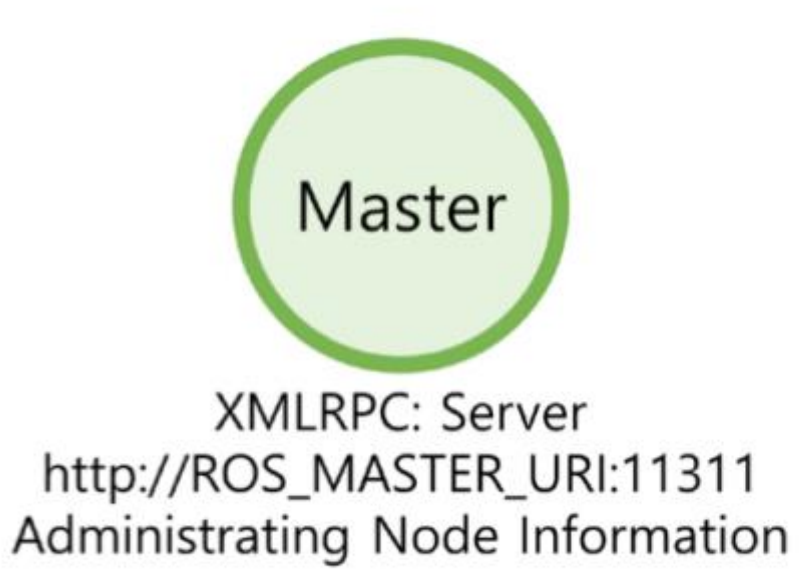
ROS

ROS master



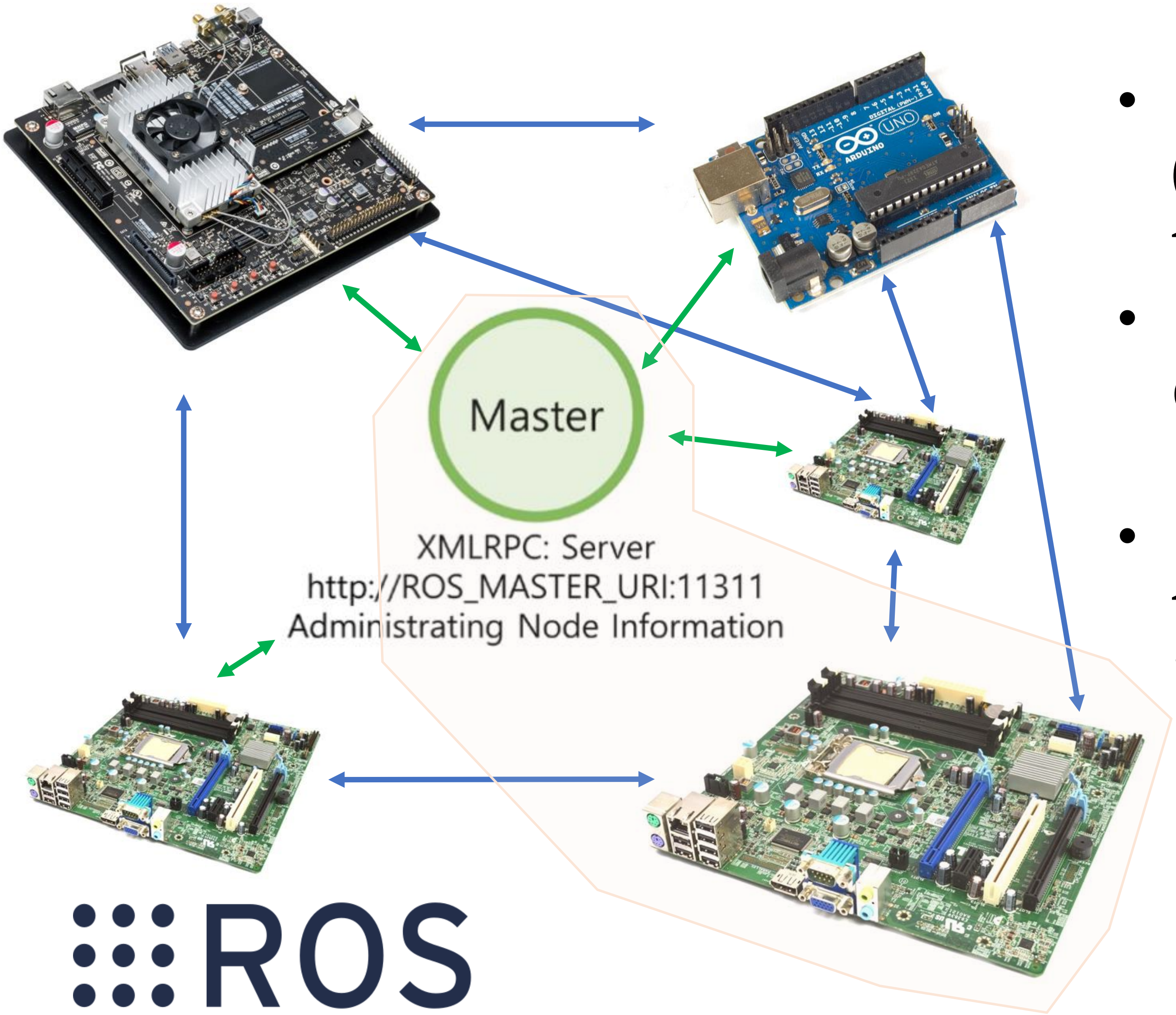
- the ROS Master provides naming and registration services to the rest of the nodes in the ROS system.
- it tracks publishers and subscribers to topics.
- it enables individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

ROS master and nodes



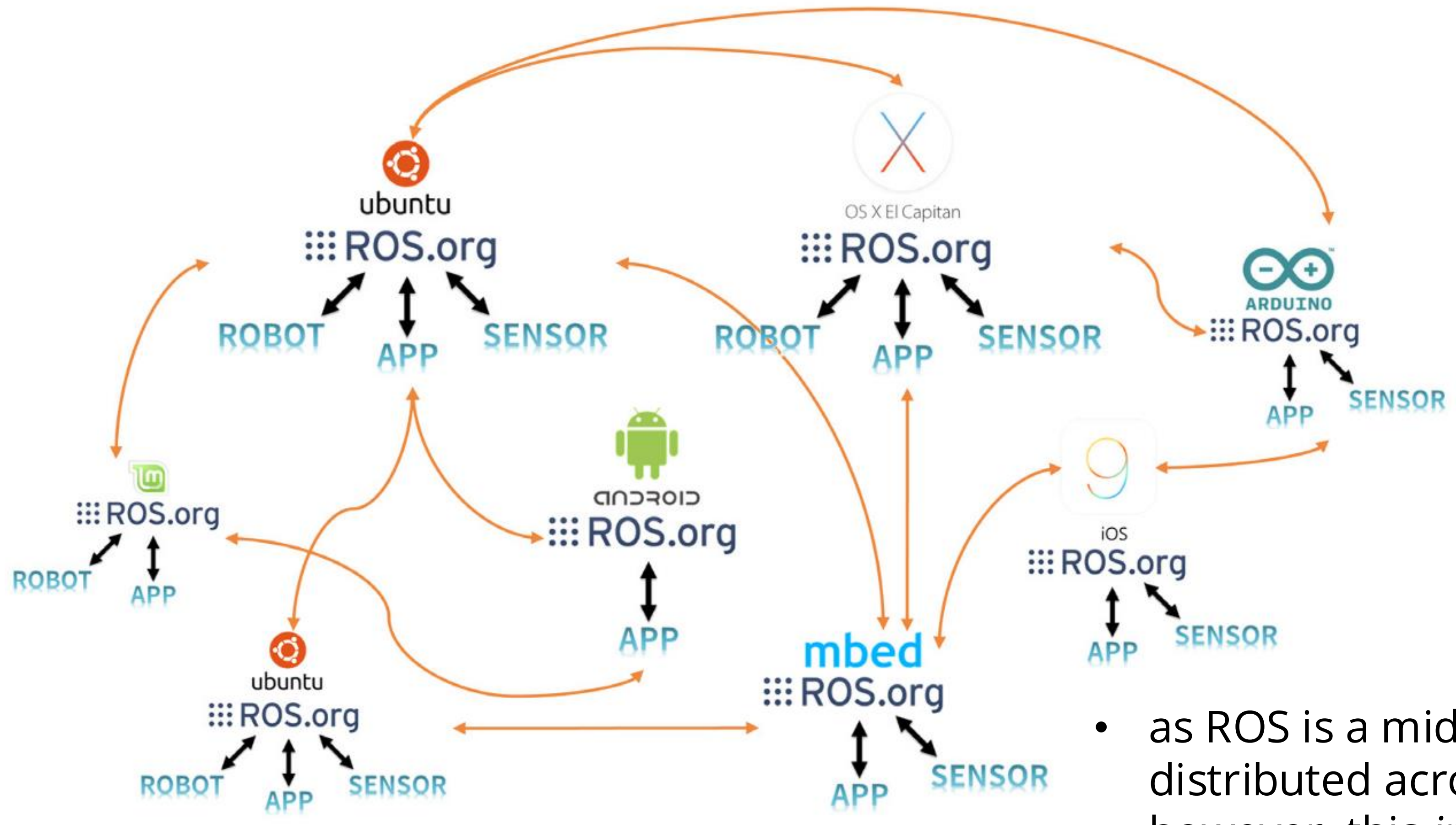
- the ROS master is a process and it is defined by its IP/port shared across all nodes
- acts as coordinator and manages the communication among nods
- this allows nodes to be distributed on different machines
  (in the same network)
- this mechanism allows to decouple the execution of a process from the machine where the process is distributed

ROS master and nodes



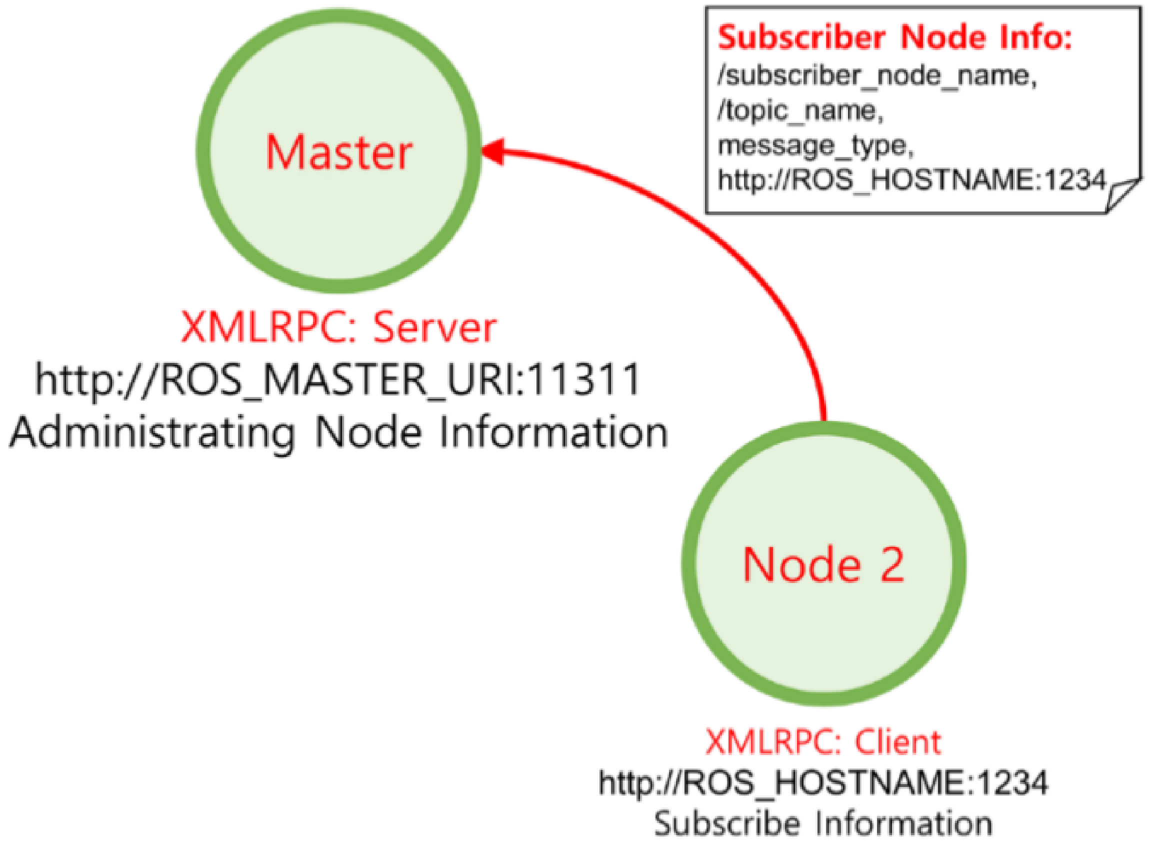
- robots may have to perform several (computationally intensive) tasks together
- hardware decoupling allows to distribute such tasks on dedicated hardware (e.g., Nvidia Jetson for GPUs)
- moreover, robots are <u>hardware</u> and this architecture allows to easily interface control boards for sensors, motors, etc.. (e.g., Arduino)

ROS on multiple platforms



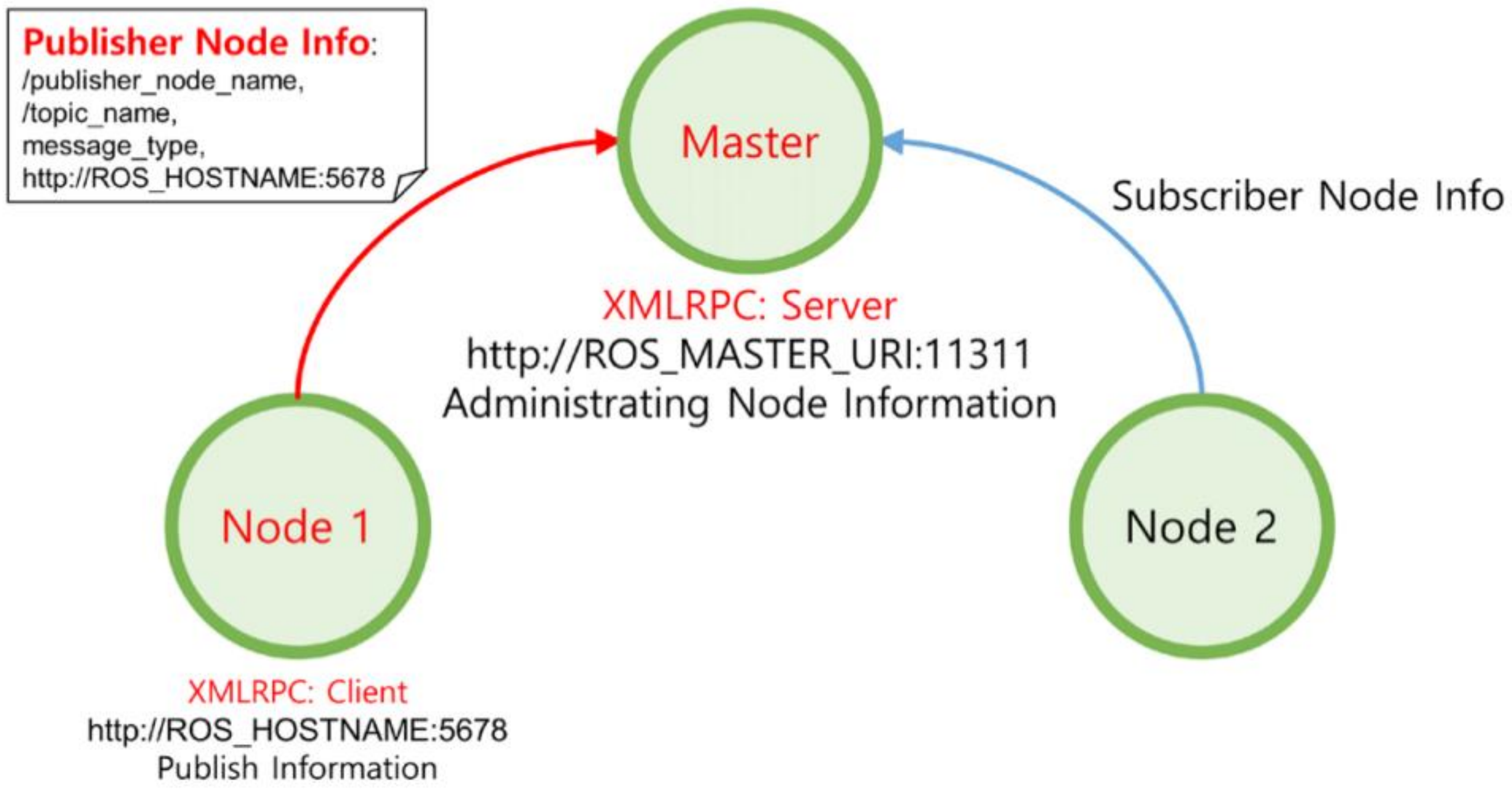
- as ROS is a middleware, computation can be distributed across different OS
- however, this *in practice* is far than ideal
- OS independence is de-facto provided for linux-based and embedded systems.
- rule-of-thumb: use Ubuntu for non-embedded systems not all versions either, but this is improved with ROS2

Set up a ROS topic publisher/subscriber



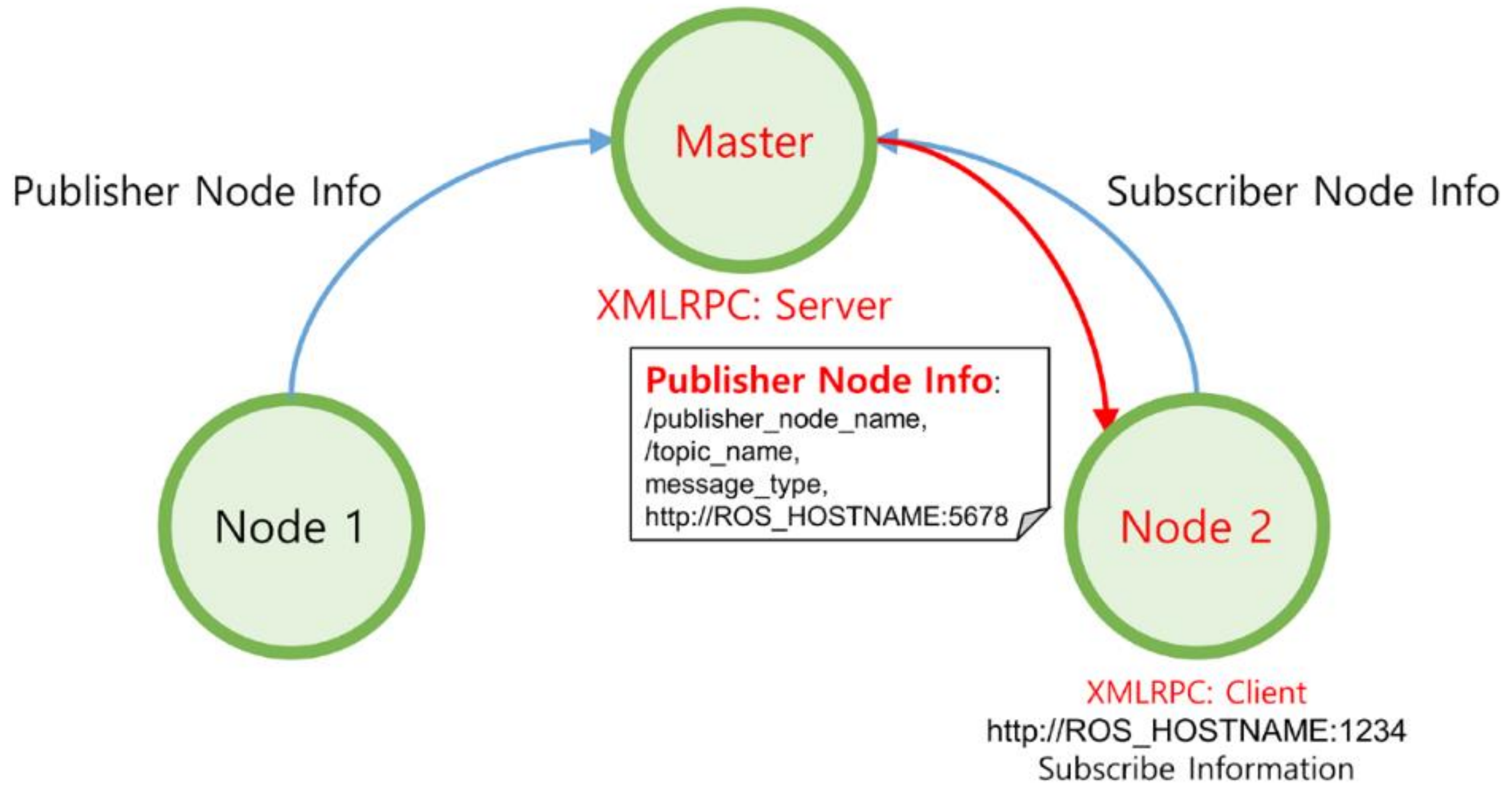
- a subscriber node registers to the ROS MASTER
- and announces its
  - Name
  - Topic name
  - Message Type
- communication is performed using XMLRPC

Set up a ROS topic publisher/subscriber



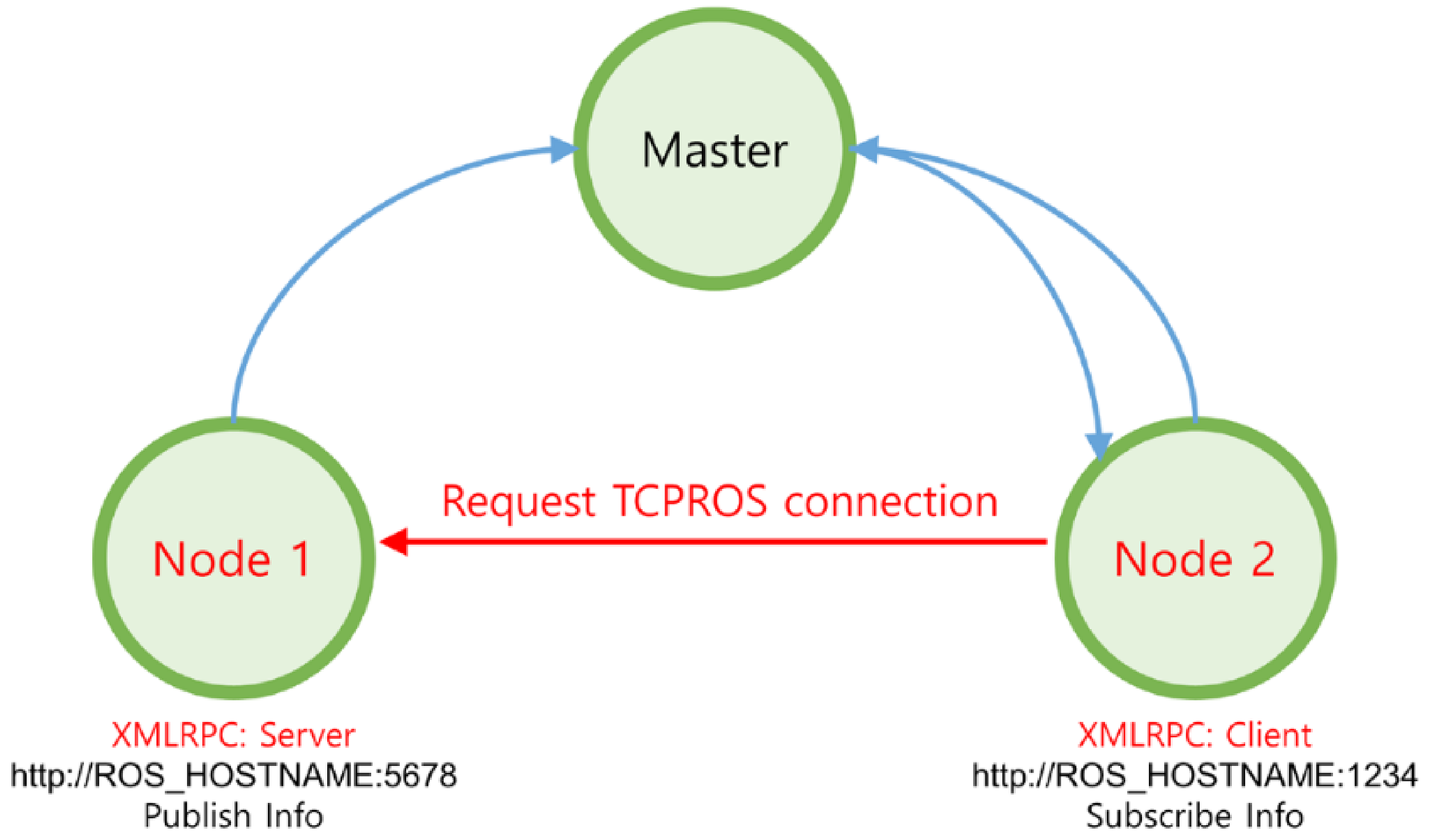
A publisher node now registers to the
ROS MASTER

Set up a ROS topic publisher/subscriber



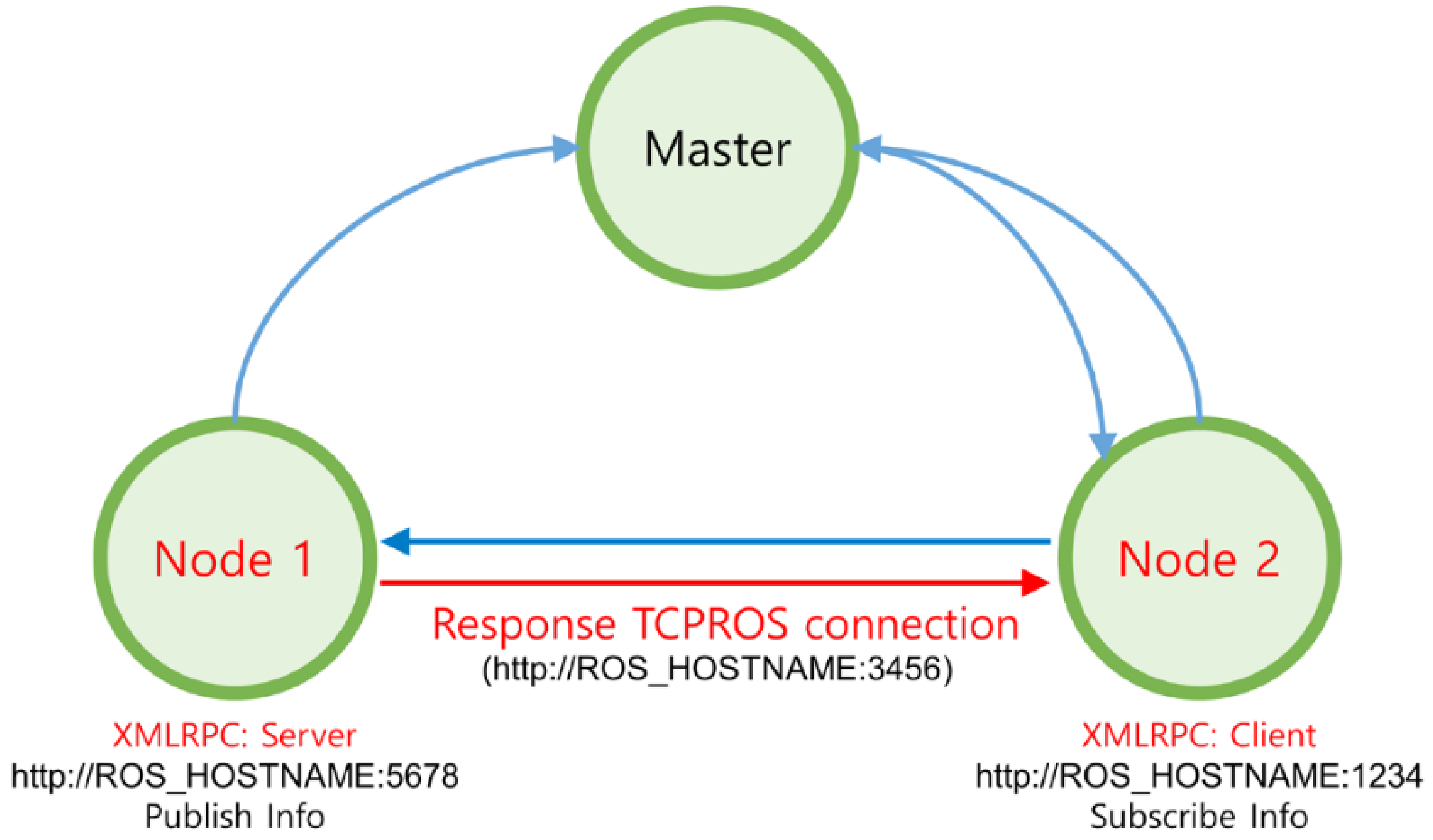
The ROS MASTER distributes info as all subscribers that want to connect to the topic and to the publisher node

Set up a ROS topic publisher/subscriber



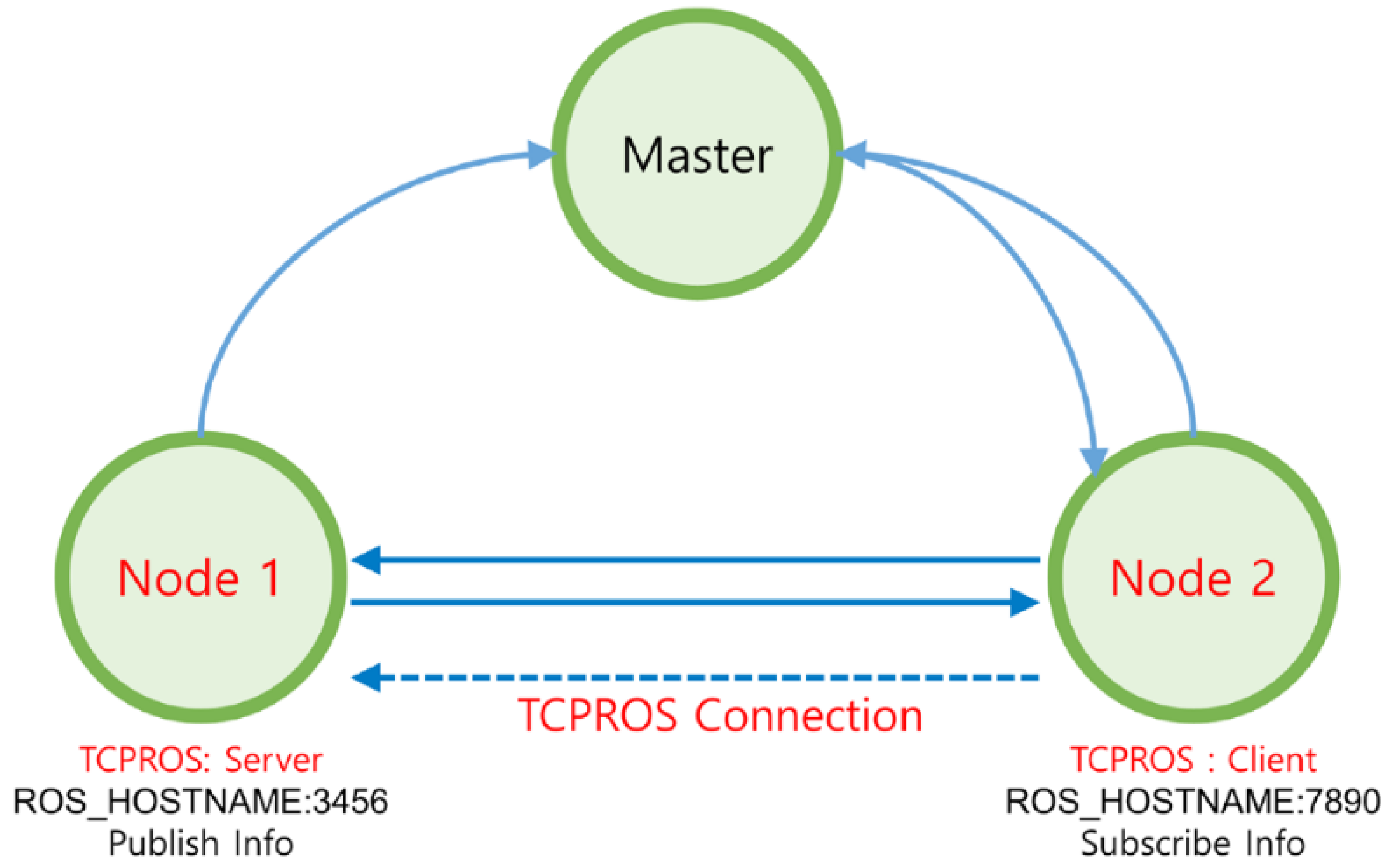
The subscriber node requests a direct connection to the published node and transmits its information to the publisher node

Set up a ROS topic publisher/subscriber



The publisher node sends the URI address and port number of its TCP server in response to the connection request.

Set up a ROS topic publisher/subscriber

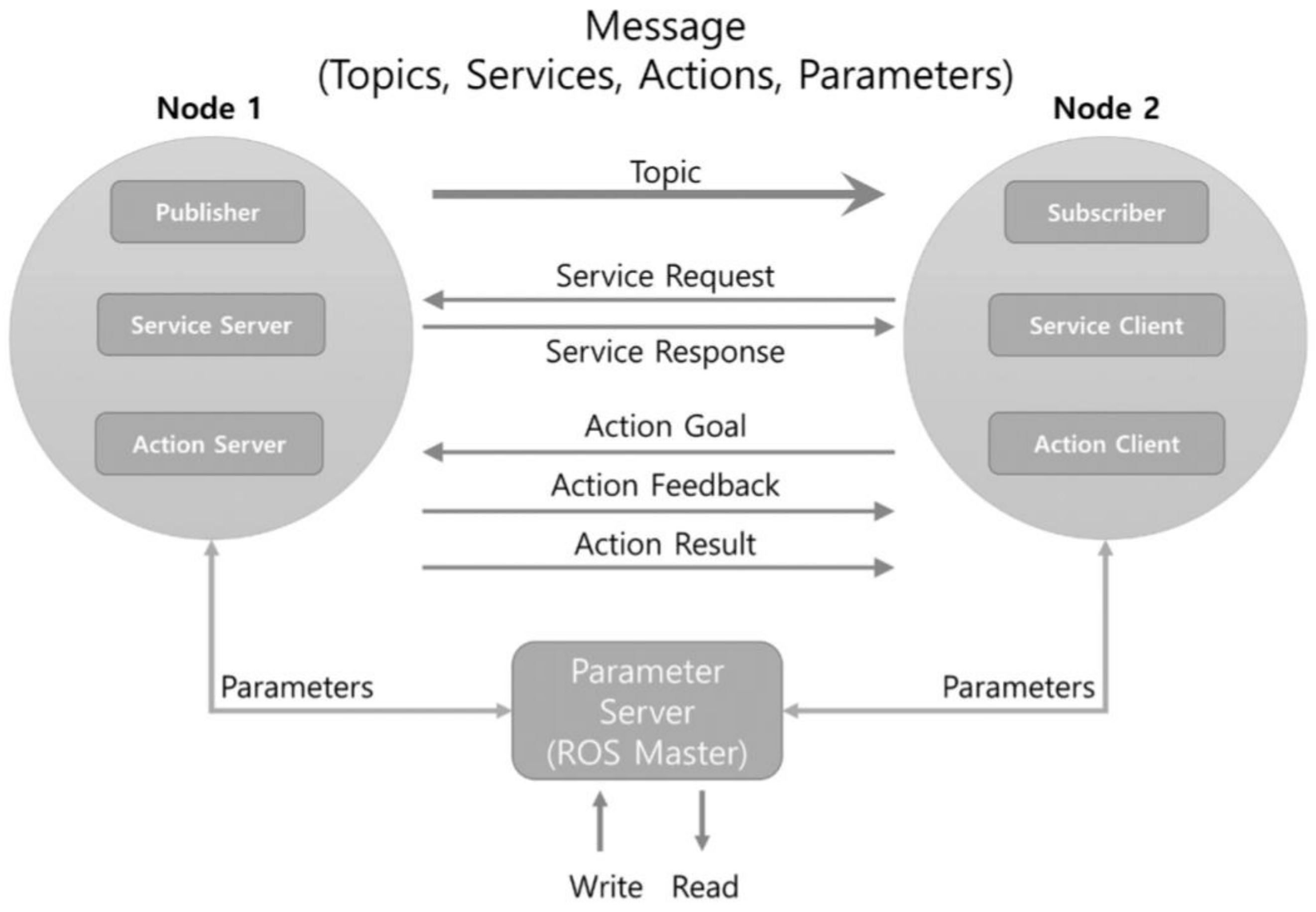


At this point a direct connection between publisher and subscriber node is established using TCPROS (TCP/IP based protocol)

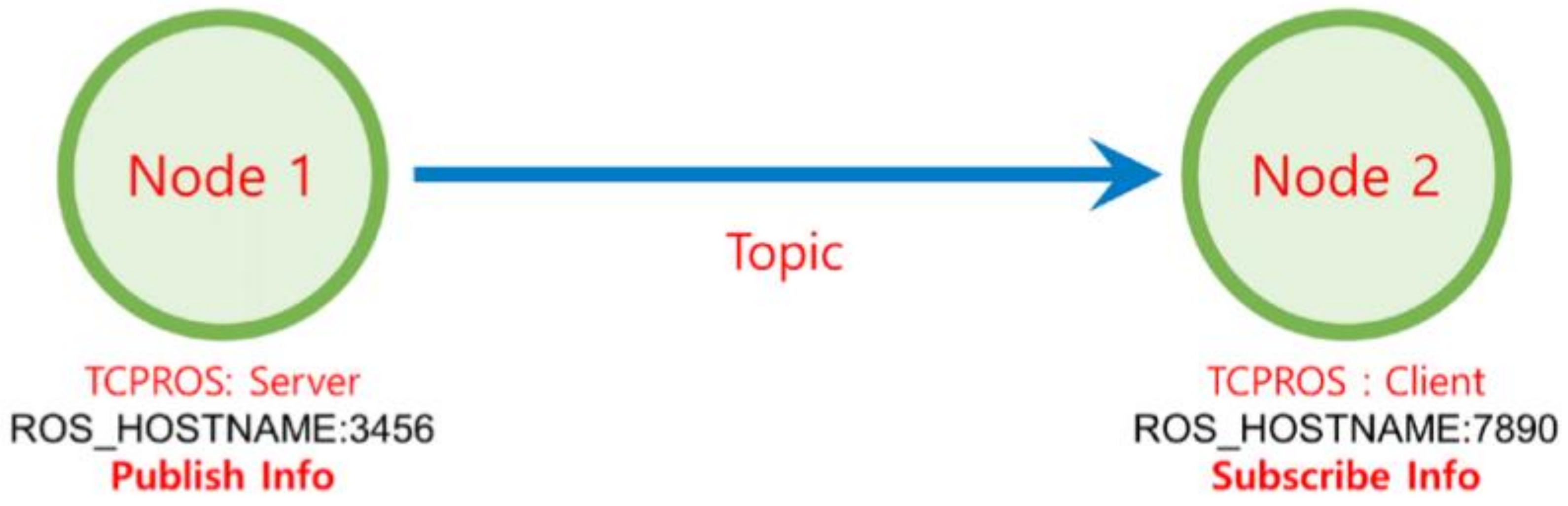
Communication among nodes

After communication between nodes is established, ROS provides 3 types of interactions

- Topics
- Services
- Actions

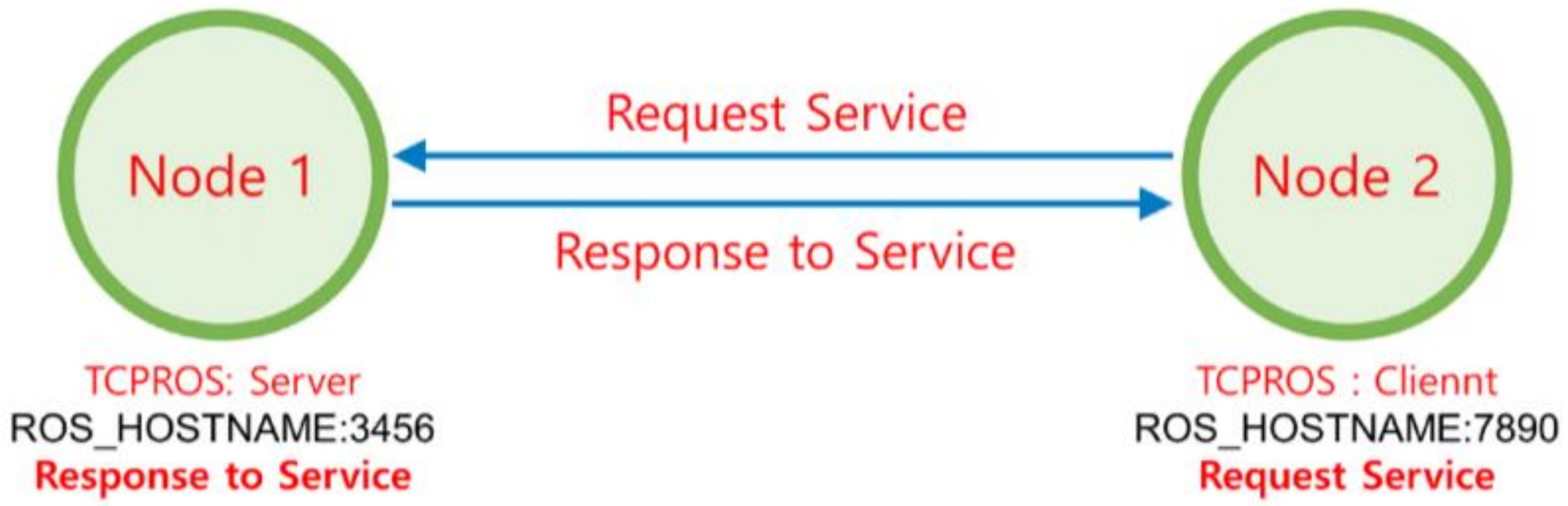


Communication among nodes



- The standard communication mechanism is using ROS topics.
- Nodes can have multiple topics
- Nodes can even use topics for internal communication
- Continuos -loop()- or one-shot (e.g. when data are ready)

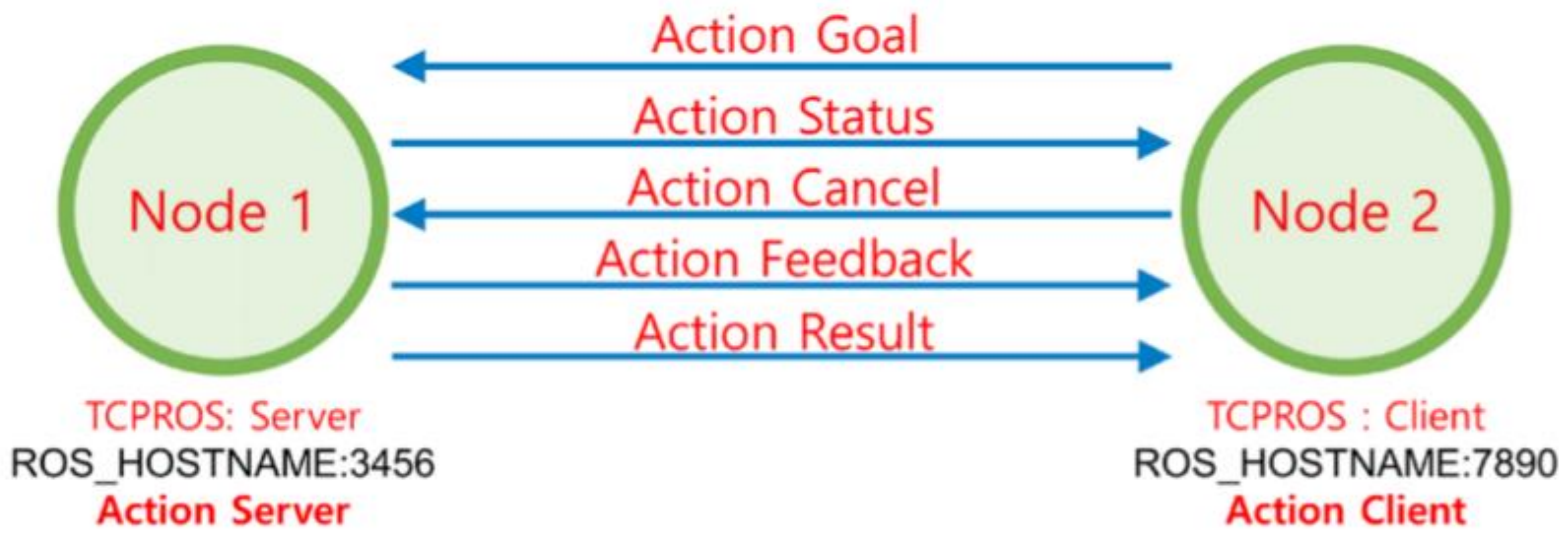
ROS *Services*



- ROS services are synchronous request from one node to another.
- Request/Reply mechanism.

A client can make a persistent connection to a service, which enables higher performance at the cost of less robustness to service provider changes.

ROS *Actions*



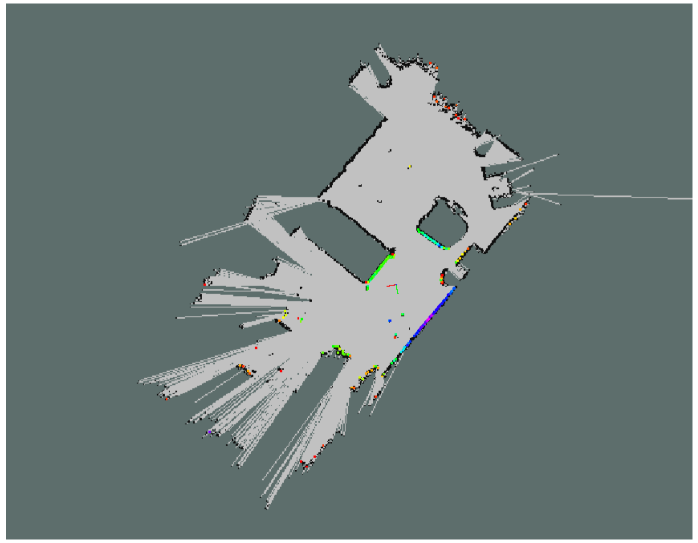
If the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.
Action Services are for these tasks.

- ROS services are asynchronous request from one node to another.
- Request/Reply mechanism, with feedbacks and the possibility to cancel the request.

ROS *parameter server*

The parameter server is a shared, multi-variate dictionary that is accessible via network APIs.

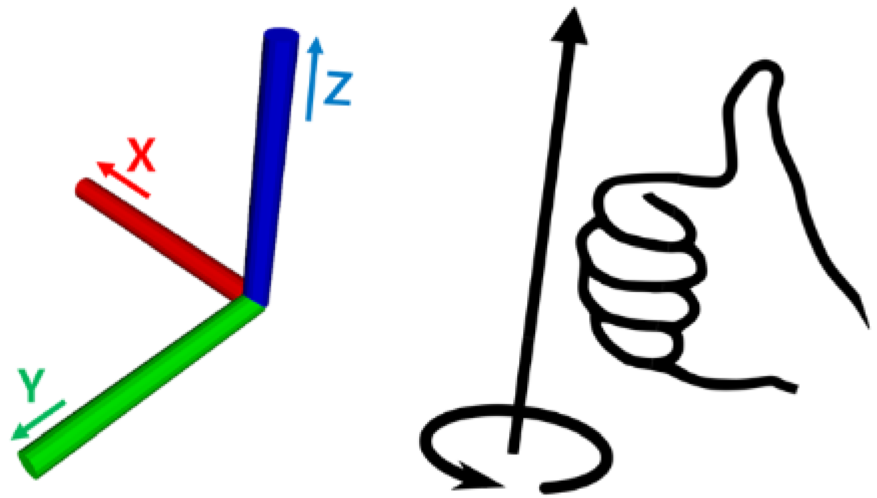
- nodes use this server to store and retrieve parameters at runtime.
- used for static, non-binary data such as configuration parameters.
- globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.



Example of params are map size/resolution and sensor configuration/settings.

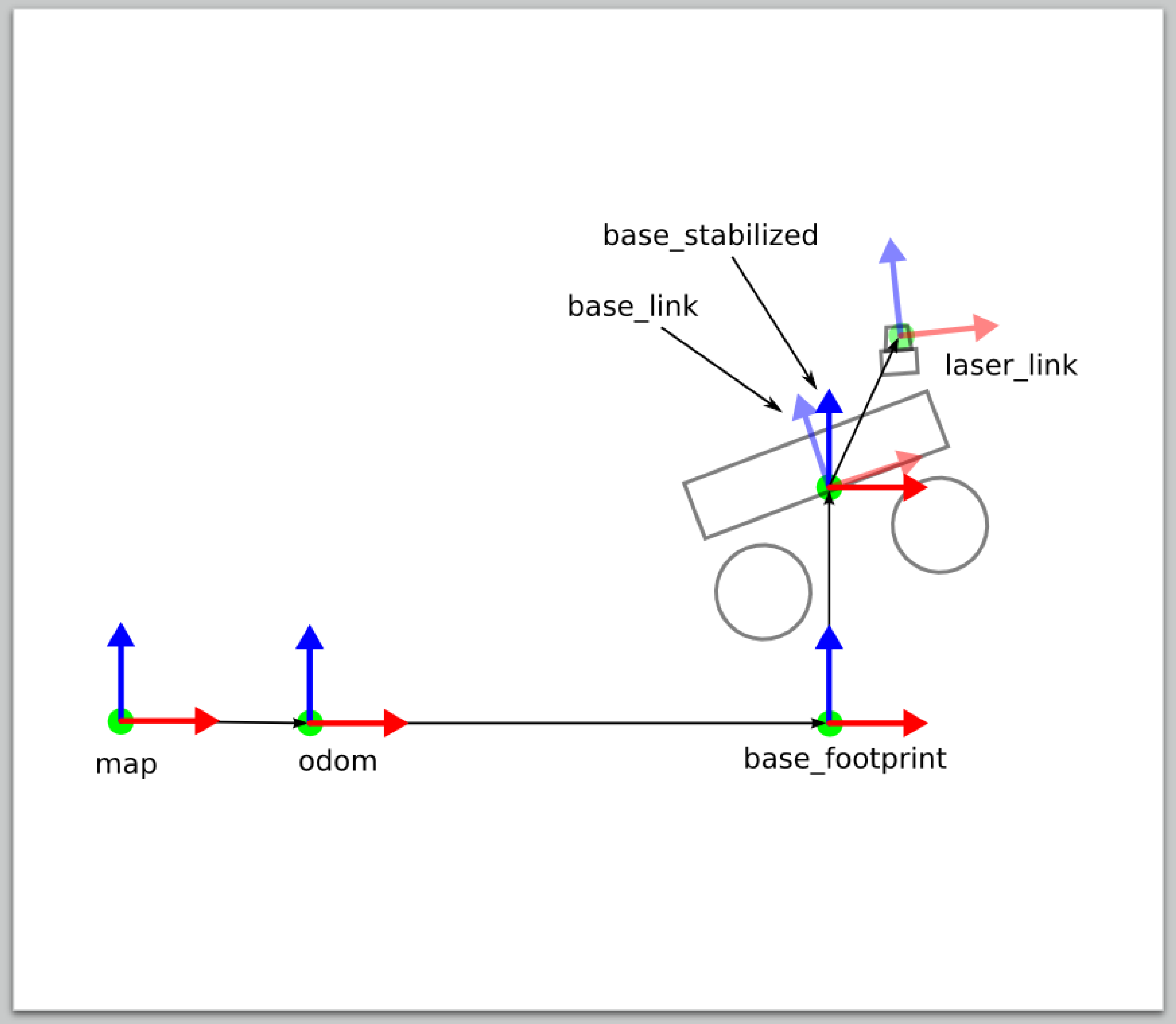
ROS *Transforms*

- in robotics programming, the robot’s joints, or wheels with rotating axes, and the position of each robot through coordinate transformation are very important
- in ROS, this is represented by TF (transforms)
- TF are published with a mechanism similar to (and parallel) the one used for ROS Topics



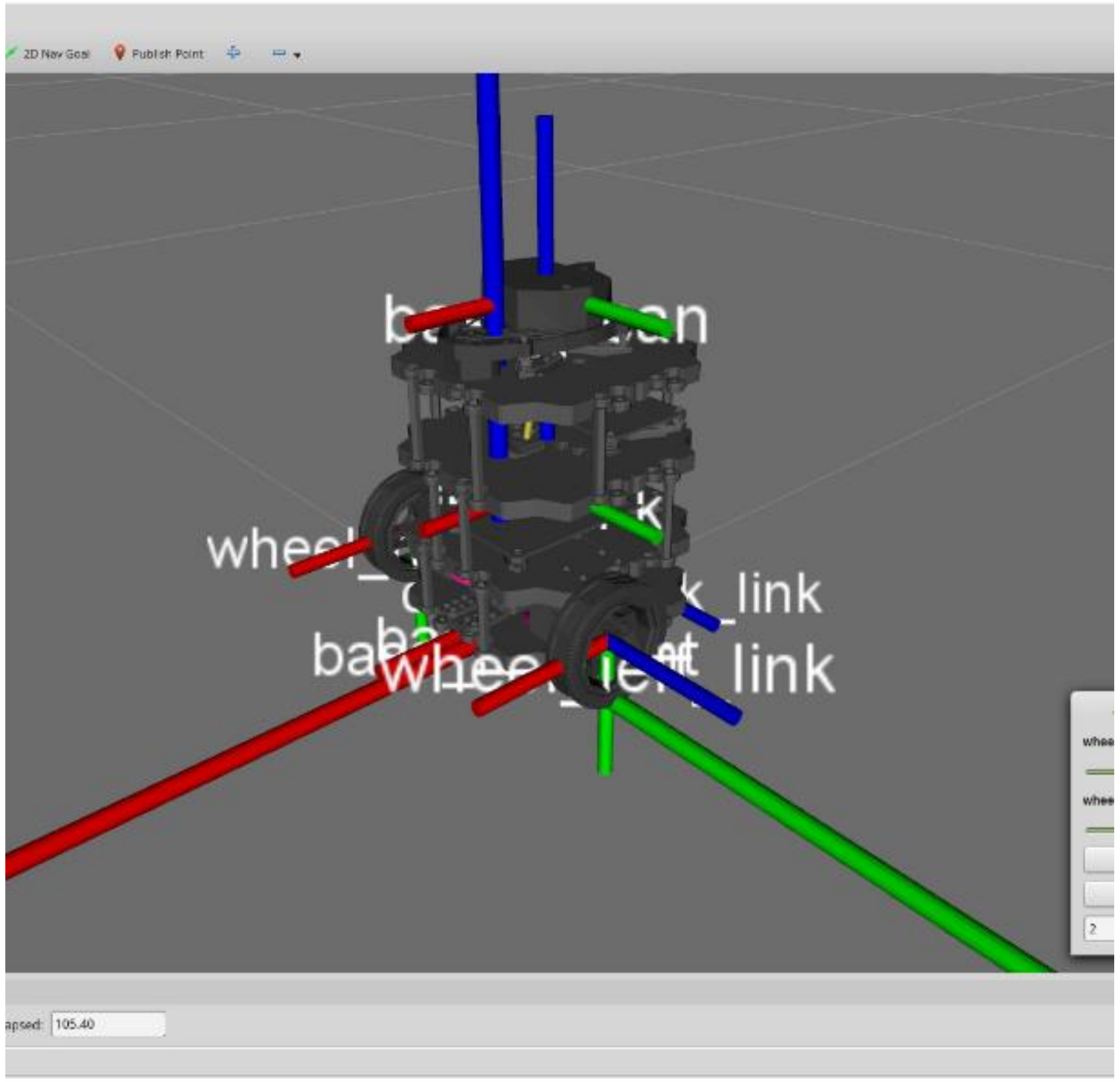
ROS *Transforms*

- all components of the robots should be connected through a chain of TF to a global reference frame (*world* or *map*)
- this is particularly important, as TFs allow the robot to project sensors onto a global reference frames



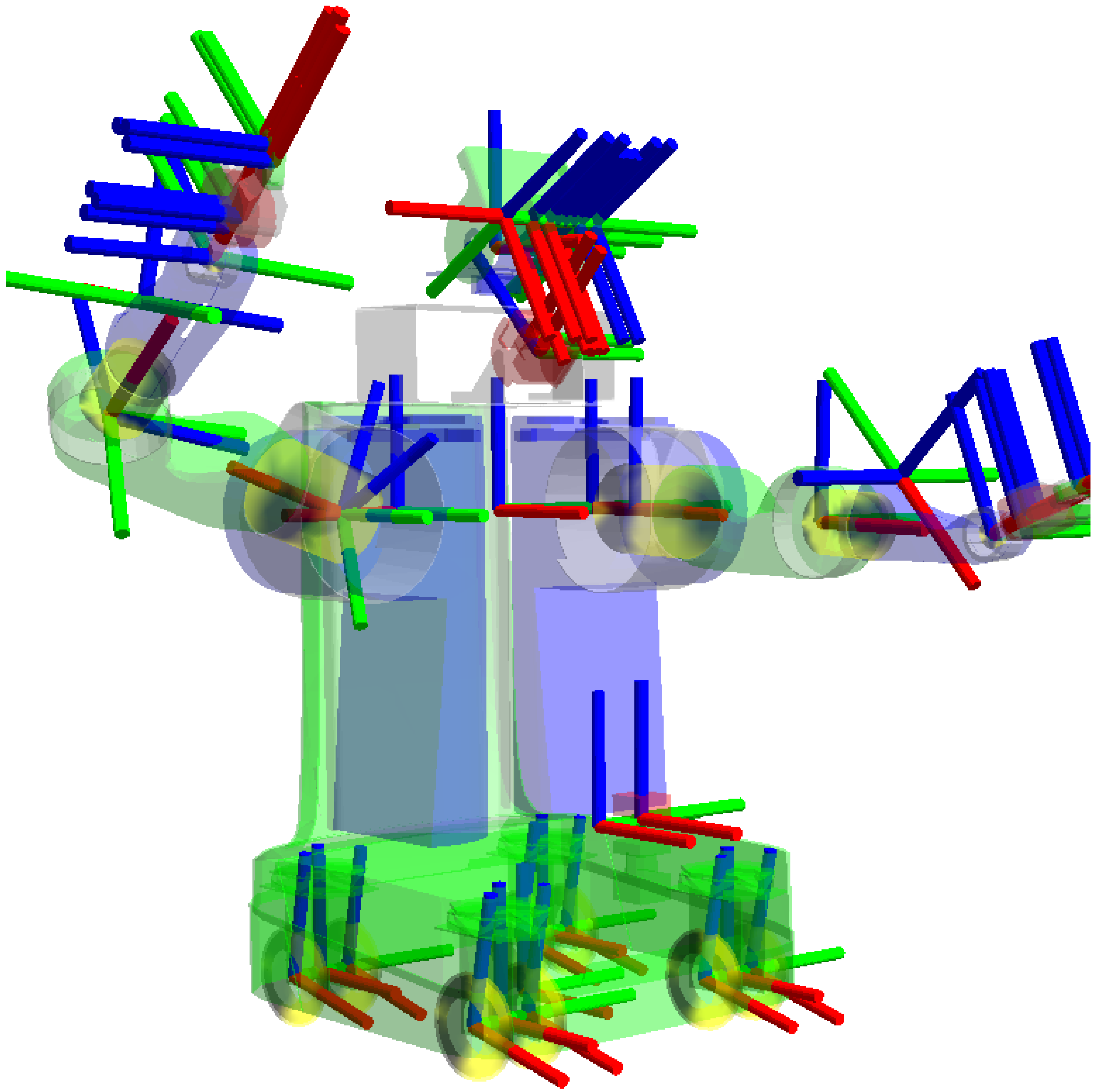
ROS *Transforms*

- some TF are static (e.g., the position of sensors w.r.t. The robot reference frame)
- some TF are dynamic and are computed real-time by nodes
  (e.g. the position of the robot in the map, the position of joints in a hand gripper)



ROS *Transforms*

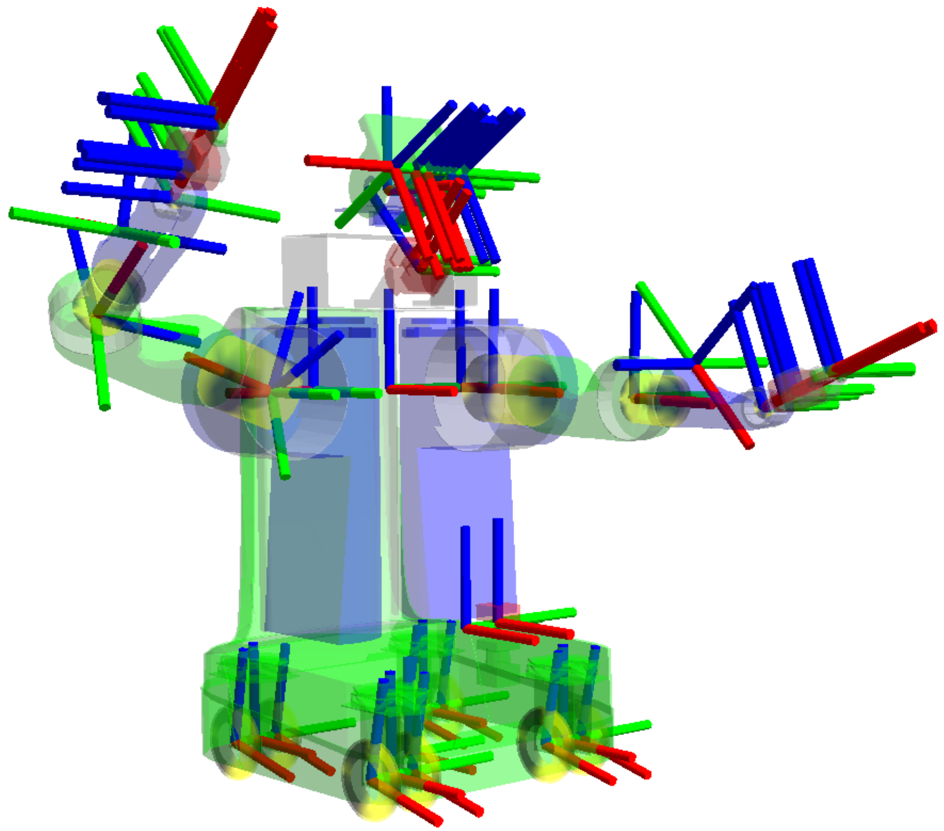
- TF can become complex, especially for robot with a lot of Degrees Of Freedom (DOF) as grippers
- ROS provides visualization tools for controlling such aspects



Developing toos

Developing a robot in ROS

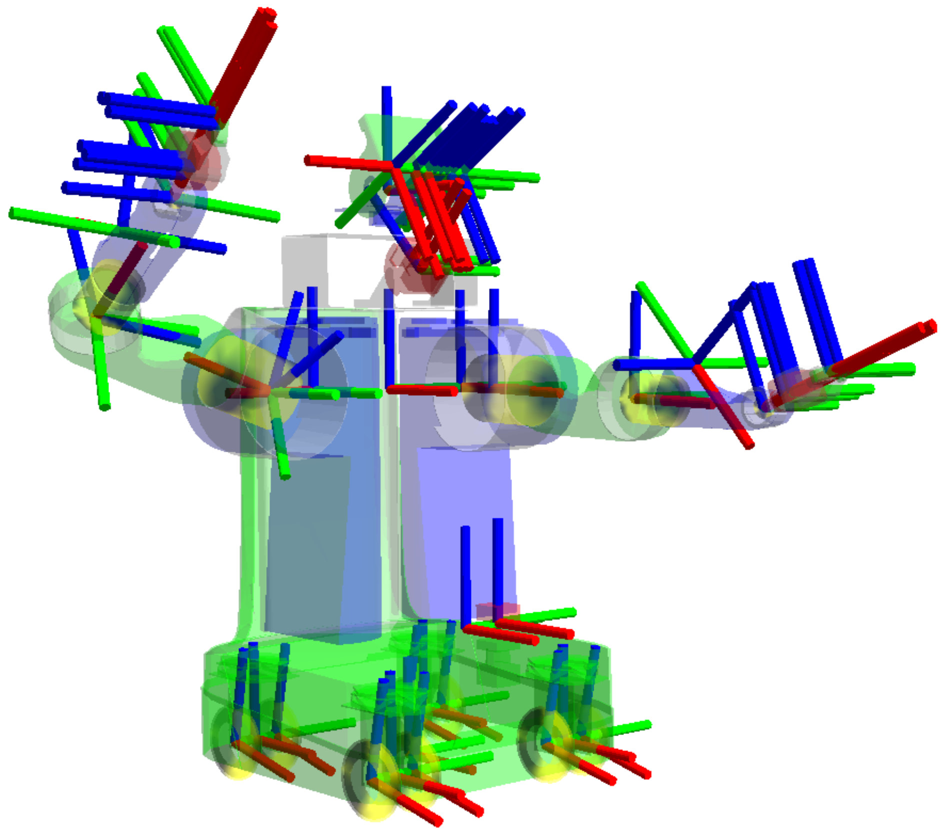
- mobile robots easily became very complex objects
- issues can emerge with single components, hardware failures, integration, …
- impossible to control all possible sources of uncertainty



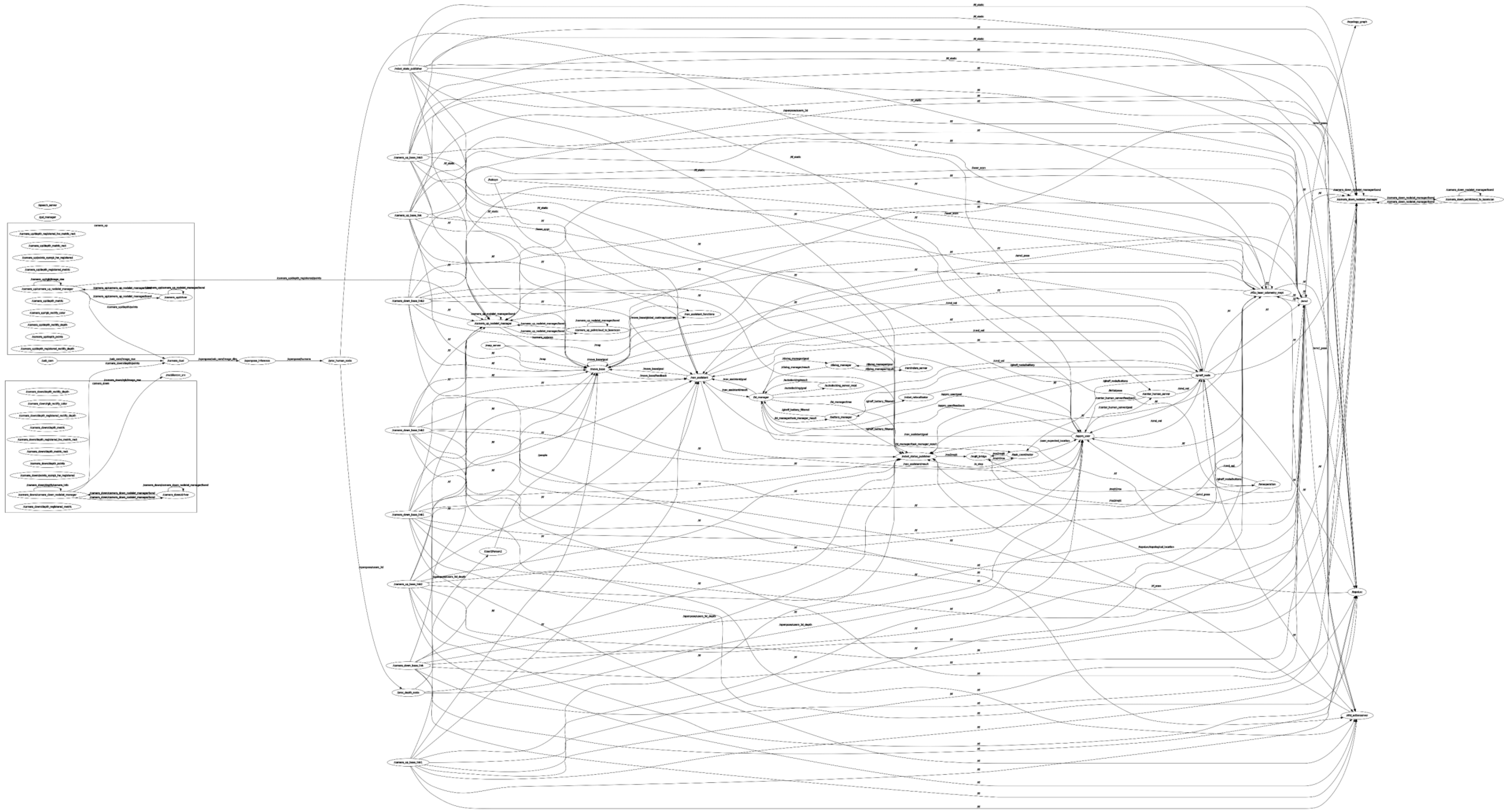
ROS

Developing a robot in ROS

- Modularity and scalability of nodes and topics help in developing complex integrated system but…
- …still the resulting ROS computational graph is impossible to be analyzed at glance



ROS



The graph of ROS nodes and topics of a real robot

How to program robots then?

- A lot of components and modules integrated among them

- Sensors and robot hardware are noisy and can fail

- Impossible to control all possible sources of uncertainty

Making even a simple run with a robot can be very time consuming

How to program robots then?

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

Developing and integrating a new functionality into a pre-existing robot can be difficult too

Why ROS is useful

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

Why ROS is useful

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

Why ROS is useful

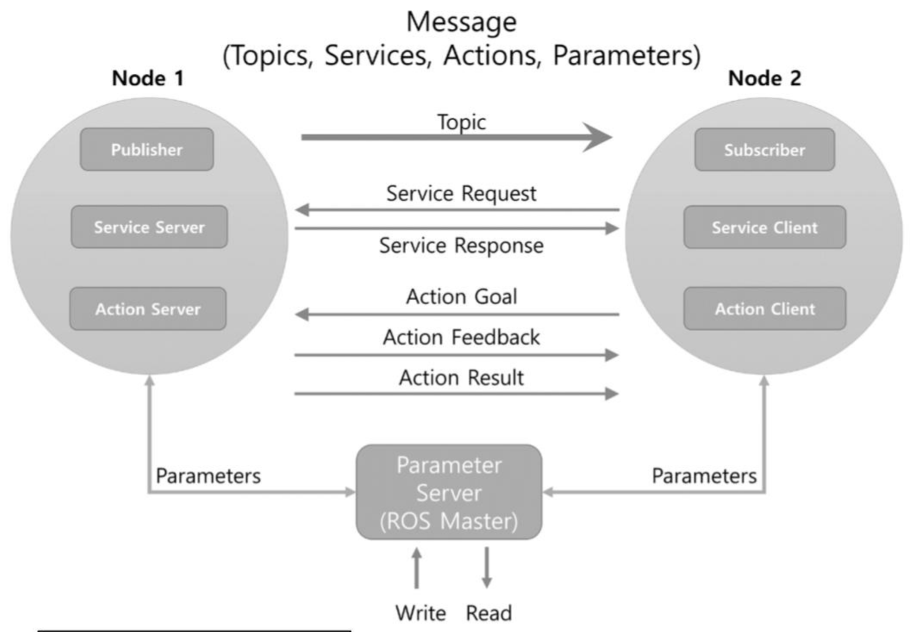
- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

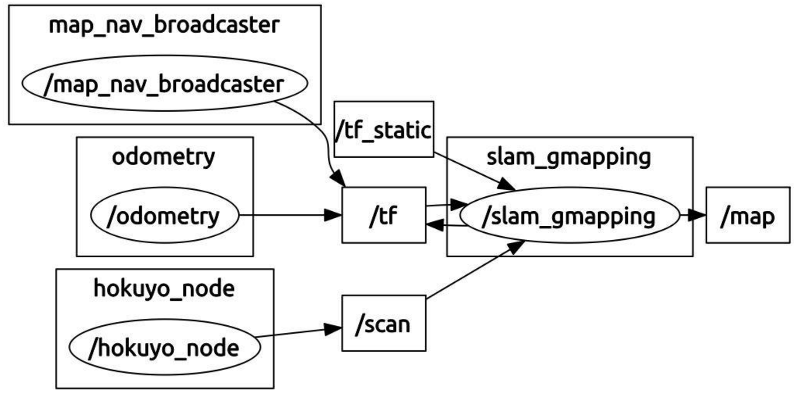
- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

ROS

ROS core concepts

- Support for all robot stack (low level motor controls to AI)
- Modularity and scalability
- Cross platform - distributed
- Repositories and code reusability
- Divide-et-impera
- Computational graph using *Nodes* and *Topics*





Why ROS is useful

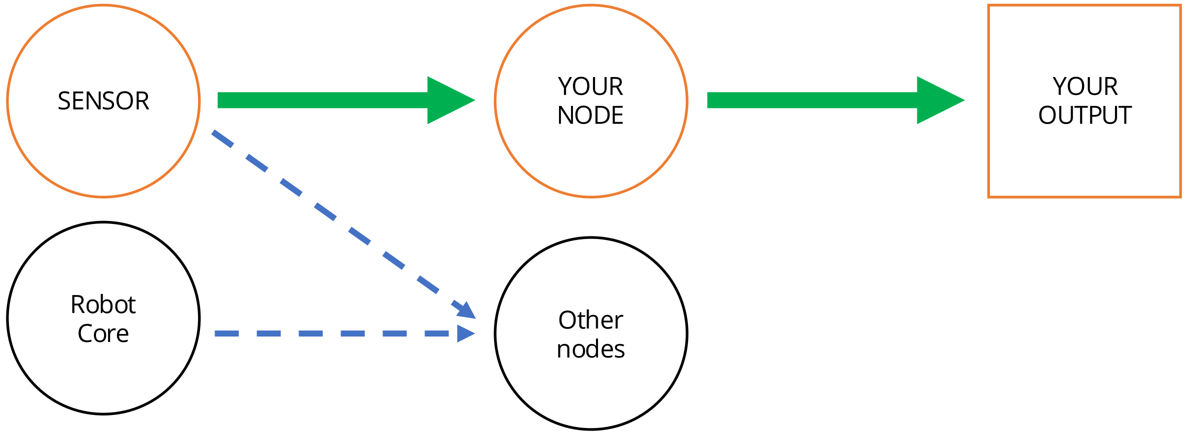
Problems

- A lot of components and module integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

Solutions

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

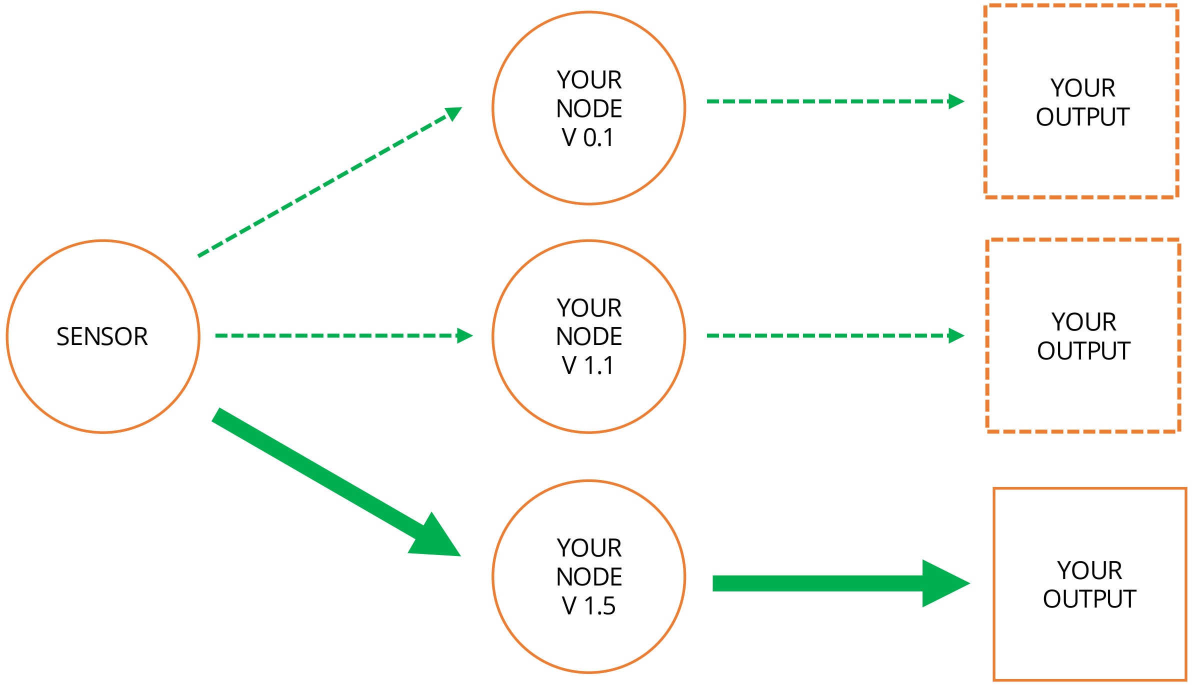
An example: writing your own *Node*



Assume that you have to implement an algorithm for a robot, e.g. a module that detects narrow passages that are challenging for the robot navigation.

ROS allows you to develop just your node while using a pre-built robot set up (from the community) and to use pre-existing robot functionalities (remote commands, mapping, odometry, sensors parsing, mapping, localization)

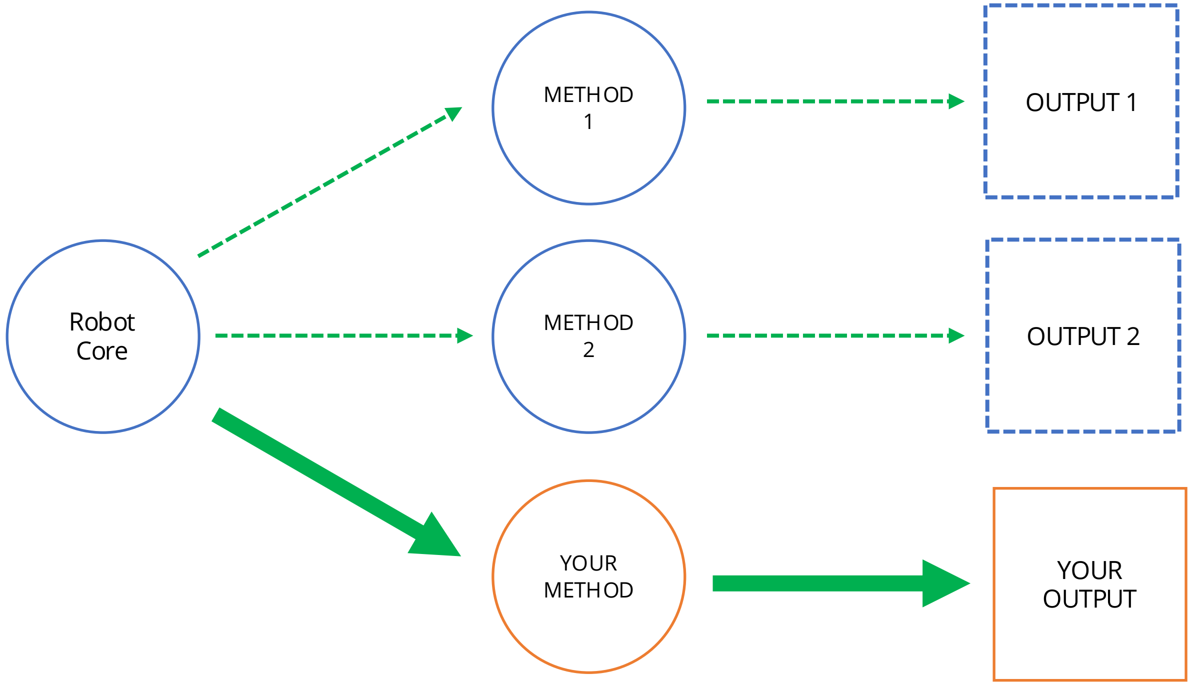
An example: writing your own *Node*



Probably you will develop several version of your node. The first one will have bugs and wont work, then a new version is produced with improvements, …

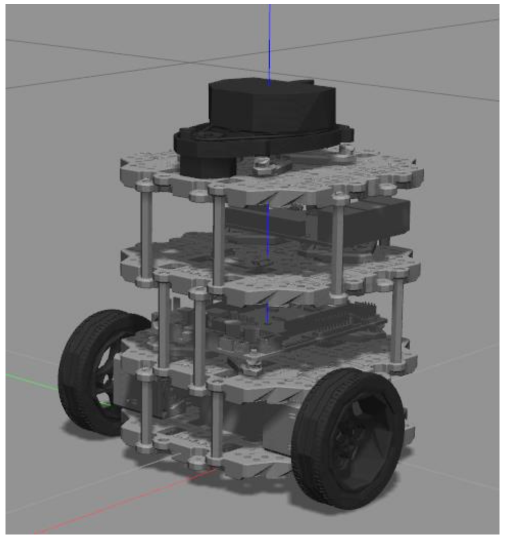
… and testing the result of different version together could be a good idea

An example: writing your own *Node*



Using *nodes* and *topics* it is also straightforward to test several methods (to see what is more useful for your robot) or to compare the results of your method with the one available to the community (and release it).

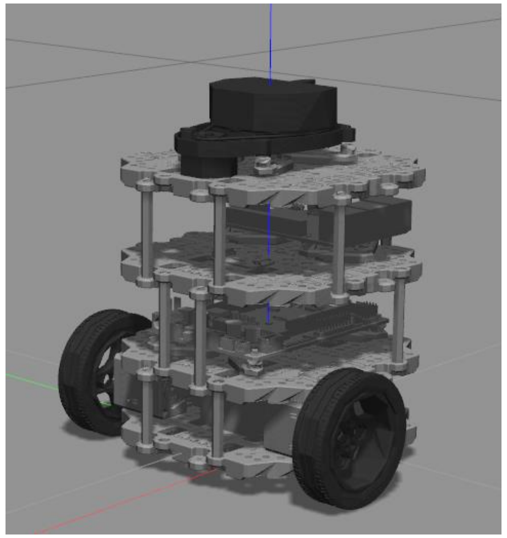
ROS and Simulations



One of the most powerful tool that ROS have is the possibility to use integrated 2D and 3D ROS simulations.

ROS simulation nodes replaces sensor drivers and allows to test the same algorithm with real robot and simulations

ROS and Simulations



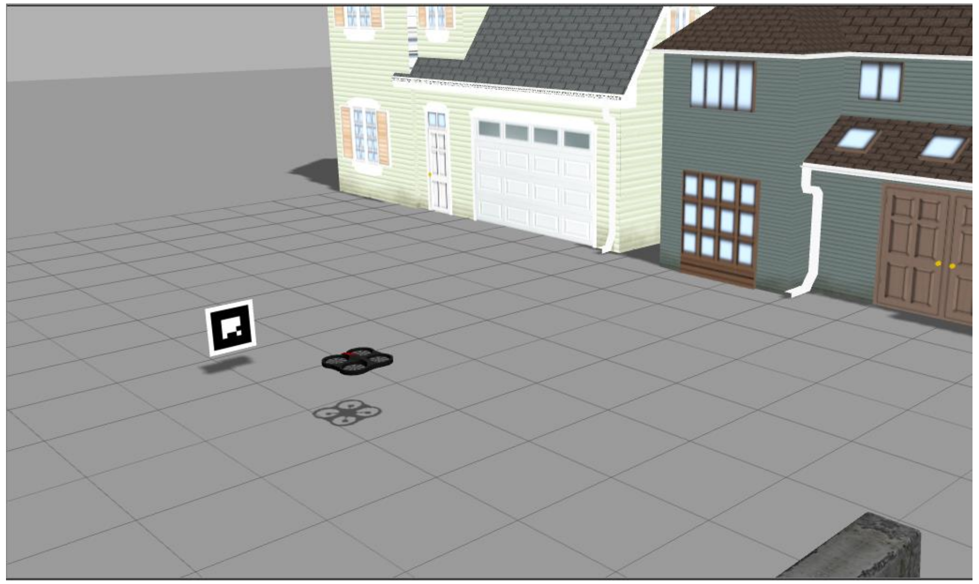
(iterative)
development of
a new method in simulations

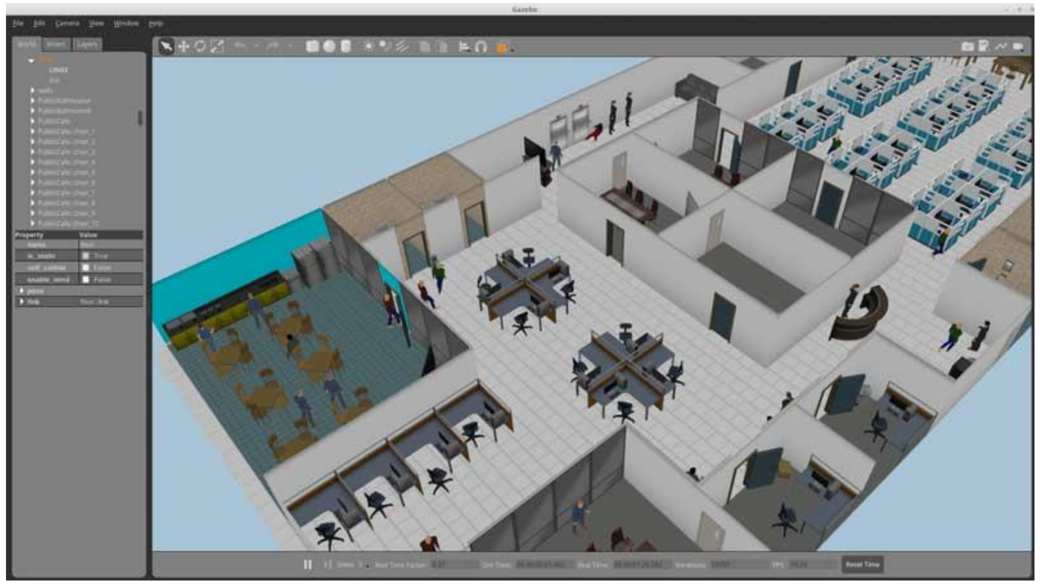
Test and validation with real robots

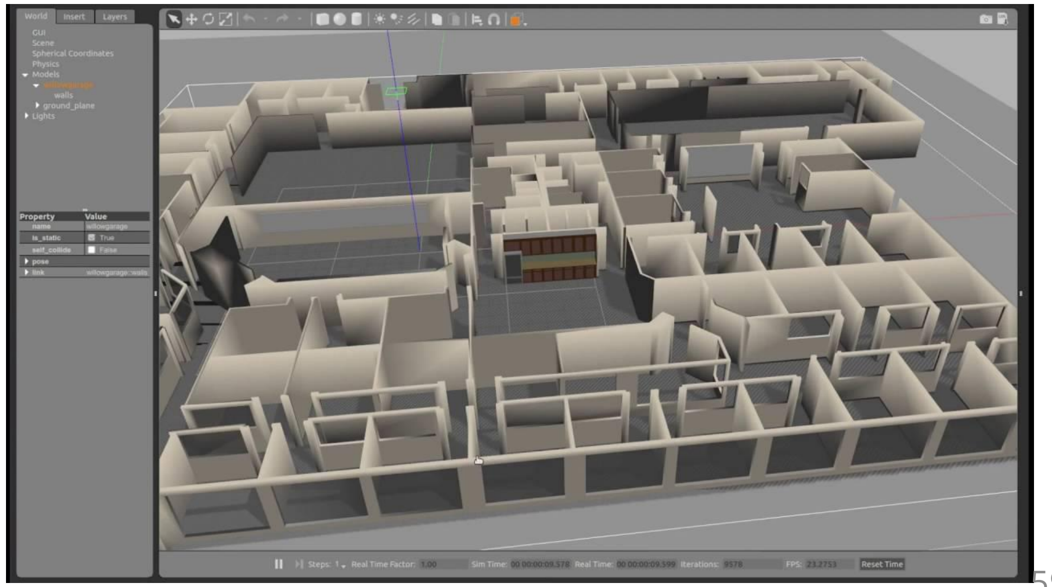
- Robots in ROS simulations are modeled starting from their real counterpart.
- This allow a fast transitioning from tests performed with simulation and with real robot just changing a few lines of code.

Simulations with Gazebo

Gazebo (3D) is the most popular and used ROS simulation tool, and it allows to simulate mobile robots, UAVs, manipulators, indoor and outdoor environments, …



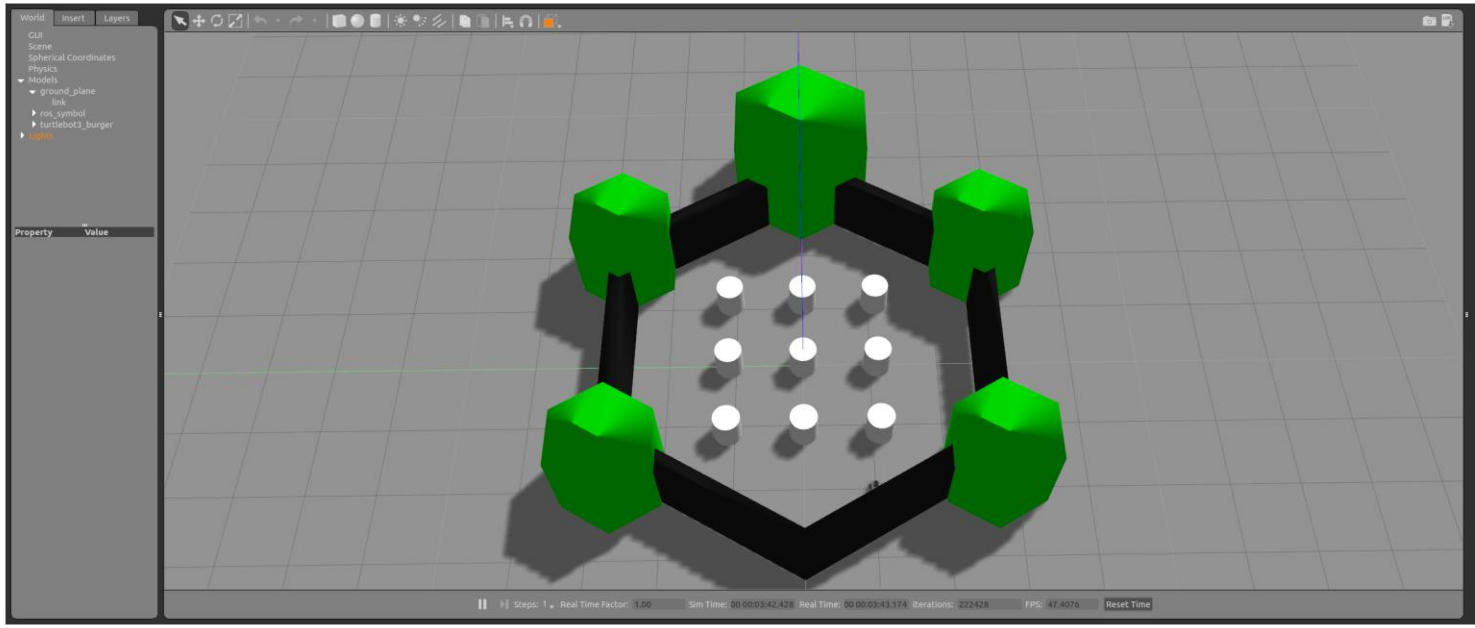
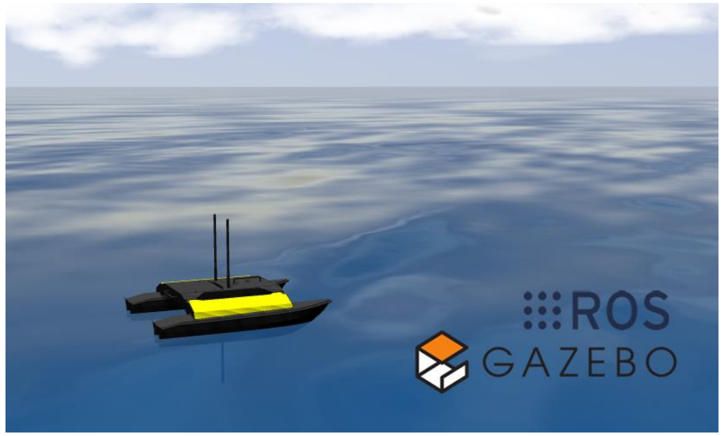


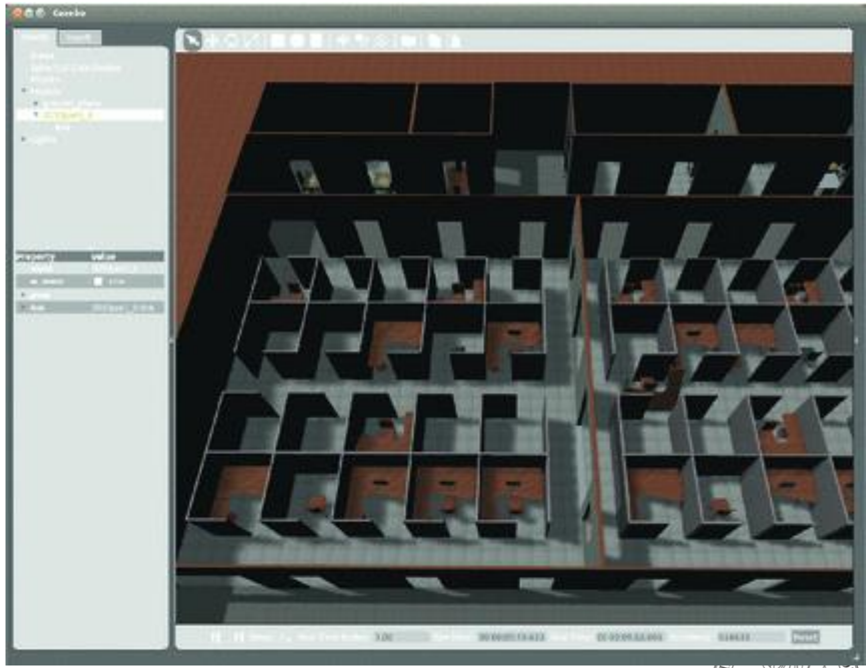


ROS

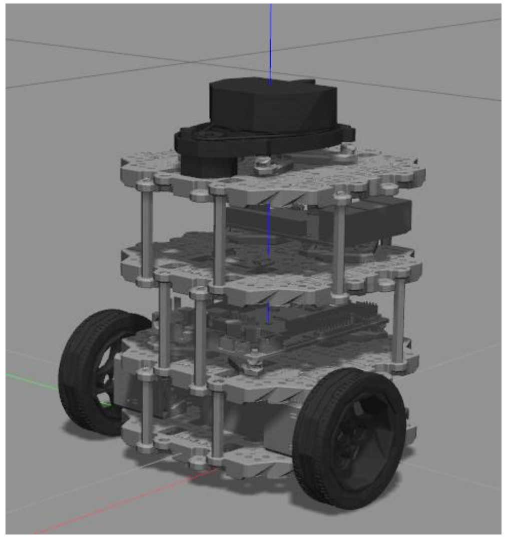
ROS Gazebo

- ROS-Gazebo is installed with ROS
- You can use publicly available *worlds* or create your own with a GUI / editor (and importing 3D models)
- You can simulate toy block-world environments and complex large-scale ones, with dynamic component
- You can simulate both mobile robot and grippers



ROS Gazebo



(iterative)
development of
a new method in simulations

Test and validation with real robots

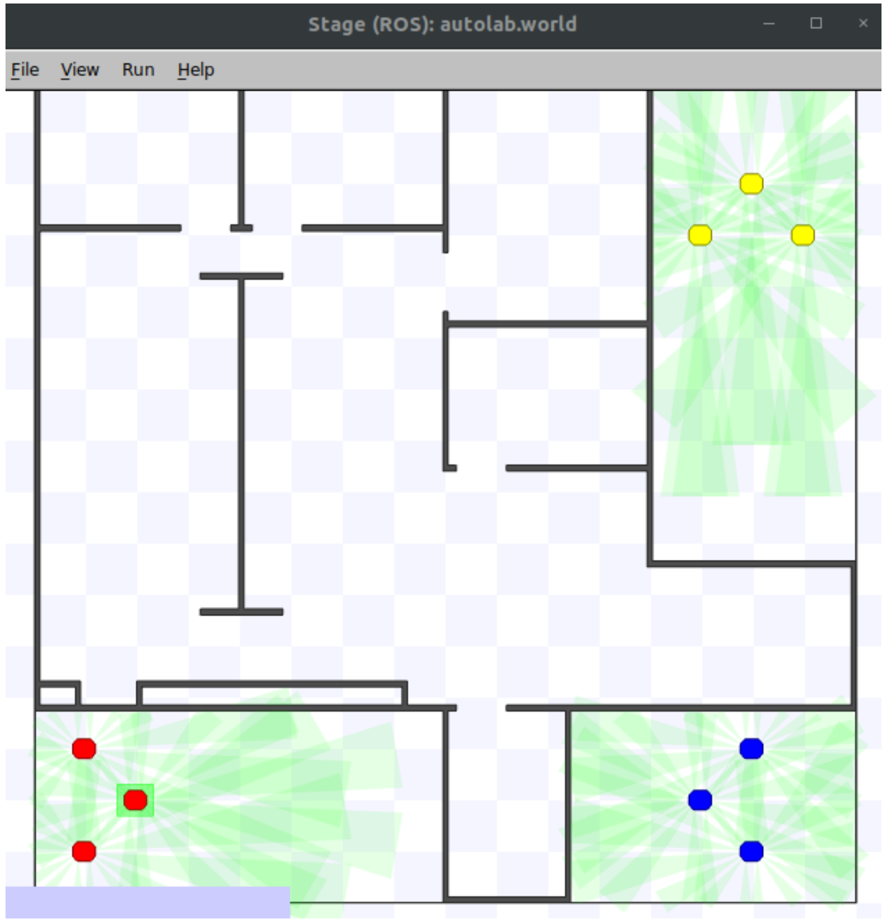
- Most robot provide a pre-configured ROS Gazebo model
- You can experiment using simulation without the need to configure the robot and the required packages and later transition to the real robot (if needed)
- They also provide documentation and tutorials

ROS

Simulations with STAGE

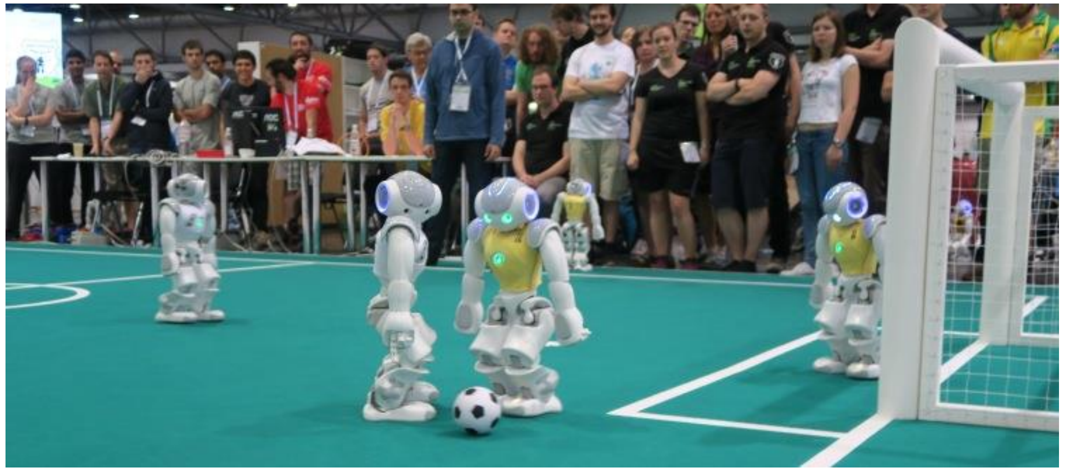
Stage is a simple 2D robot simulator.
It is useful for testing multi-robot systems, swarm robotics (even 10k robots) and for testing robotic tasks that require a higher level of abstraction.

Besides Gazebo and Stage, ROS can work with many commercial and open-source robotic simulation tools.



From simulations to reality

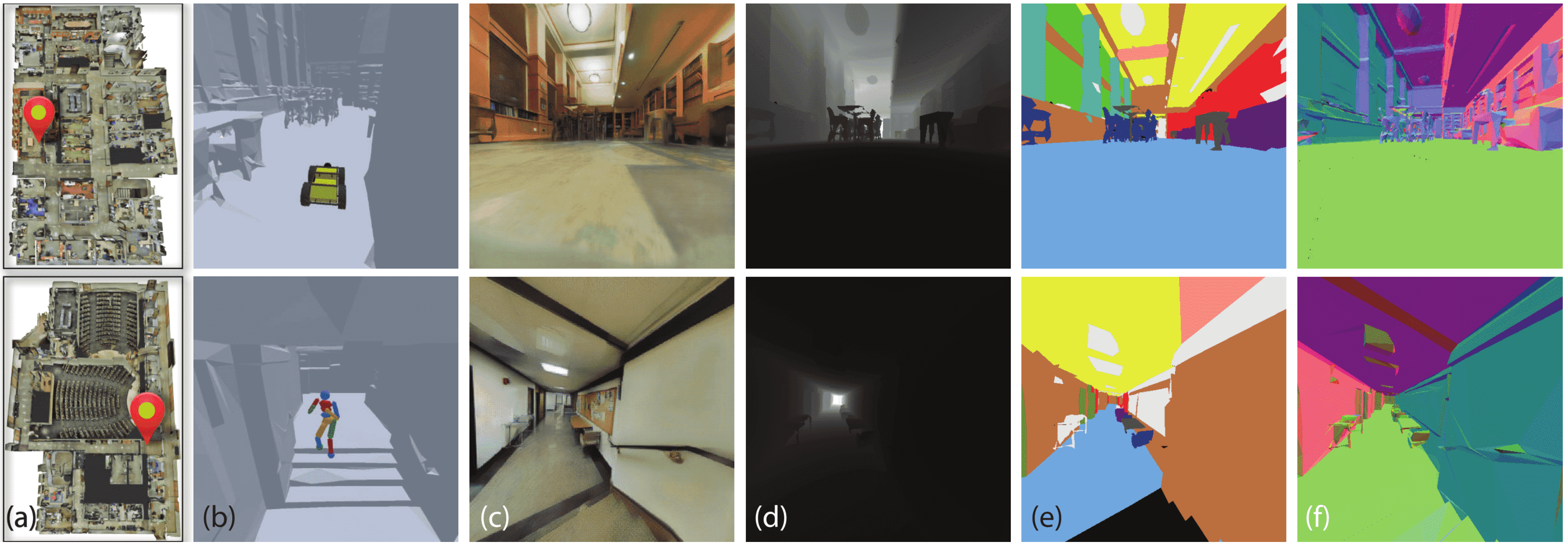
- Simulations are currently used to test complex robot behaviours
- The more robots have physical limitations (e.g. humanoid), the more simulations are used
- However, how to transfer capabilities trained in simulations to real-world-robots is still an open question

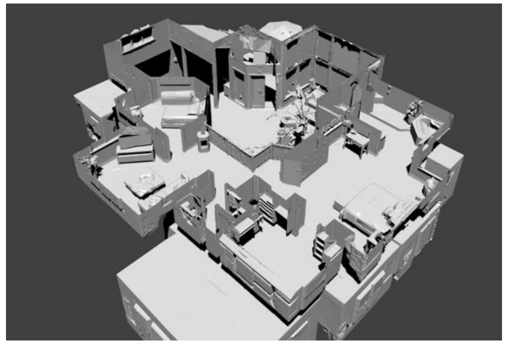


ROS

Realism in simulation

- Simulated environments look different from real ones
- Can we train/test vision-based methods in simulation? (online)
- Solution: use visually realistic simulation from real-world model

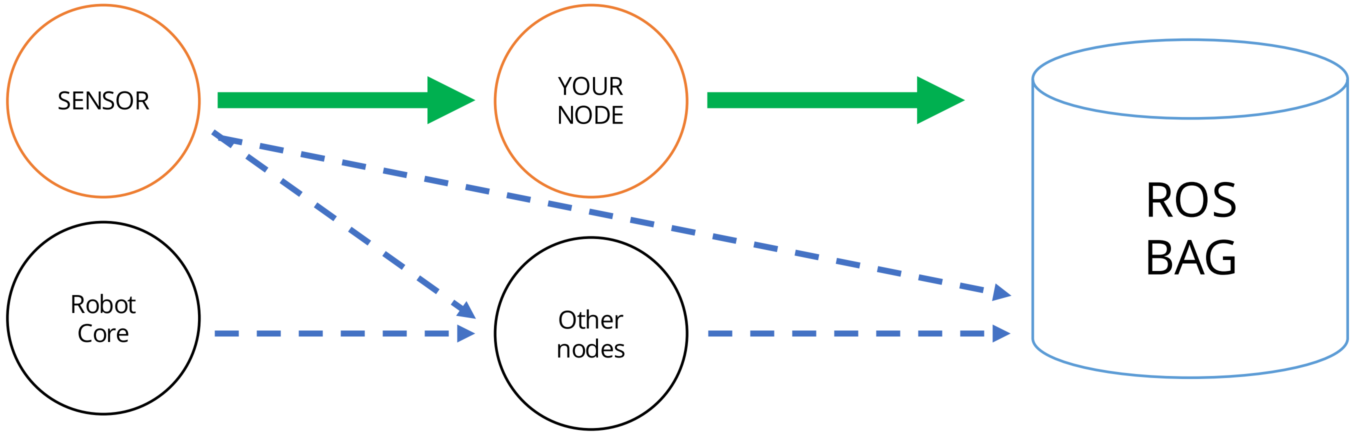




http://gibsonenv.stanford.edu/ with ROS

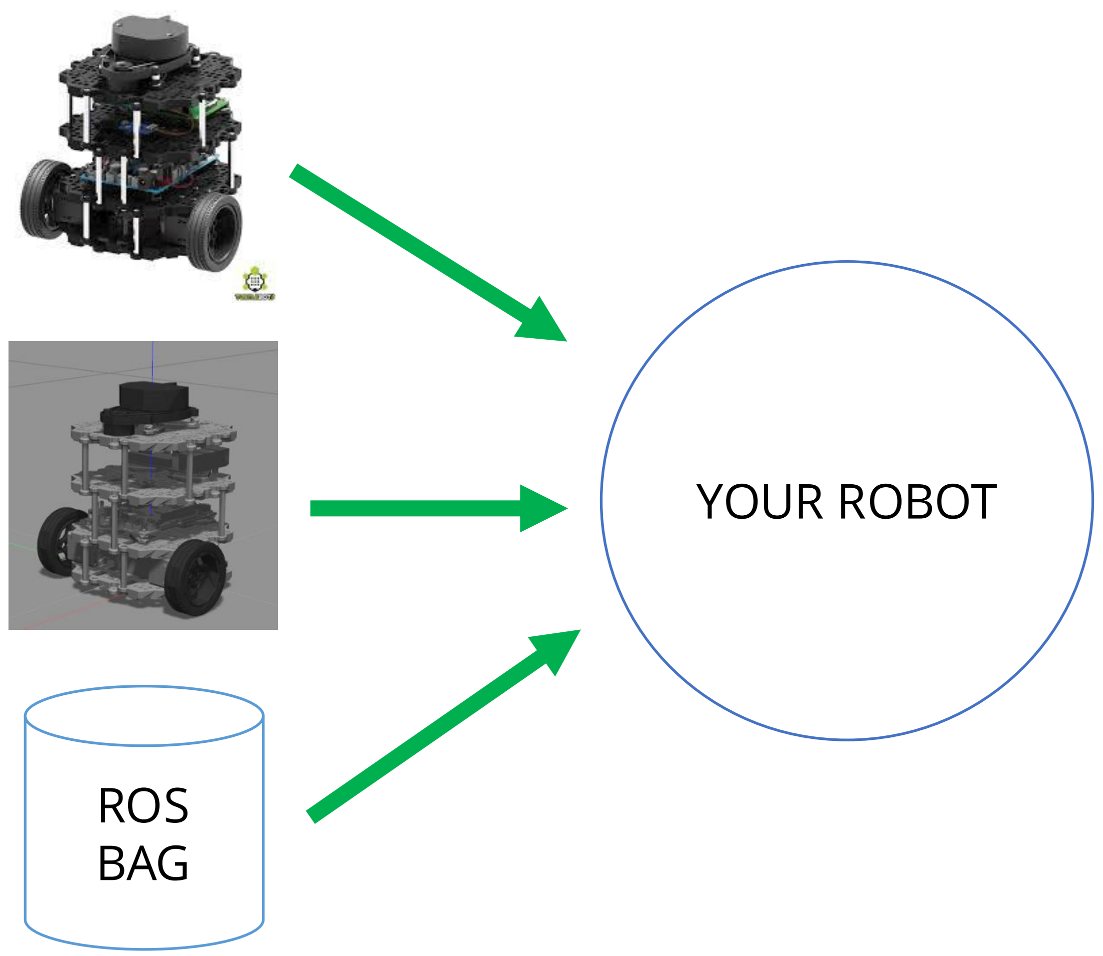
ROS

Another solution – use datasets



- Another important tool embedded in ROS is the possibility to record robot runs (in simulations or with real robots and to replay them).
- **ROS bags** store a time-stamped serialized version of all selected topics (sensors, nodes outputs, ...)
- Different algorithms can be tested with the same input to test improvements
- Bags can be reproduced at 2x, 4x, 5x → speed up of development times
- Publicly available datasets are shared among the community

Wrapping up



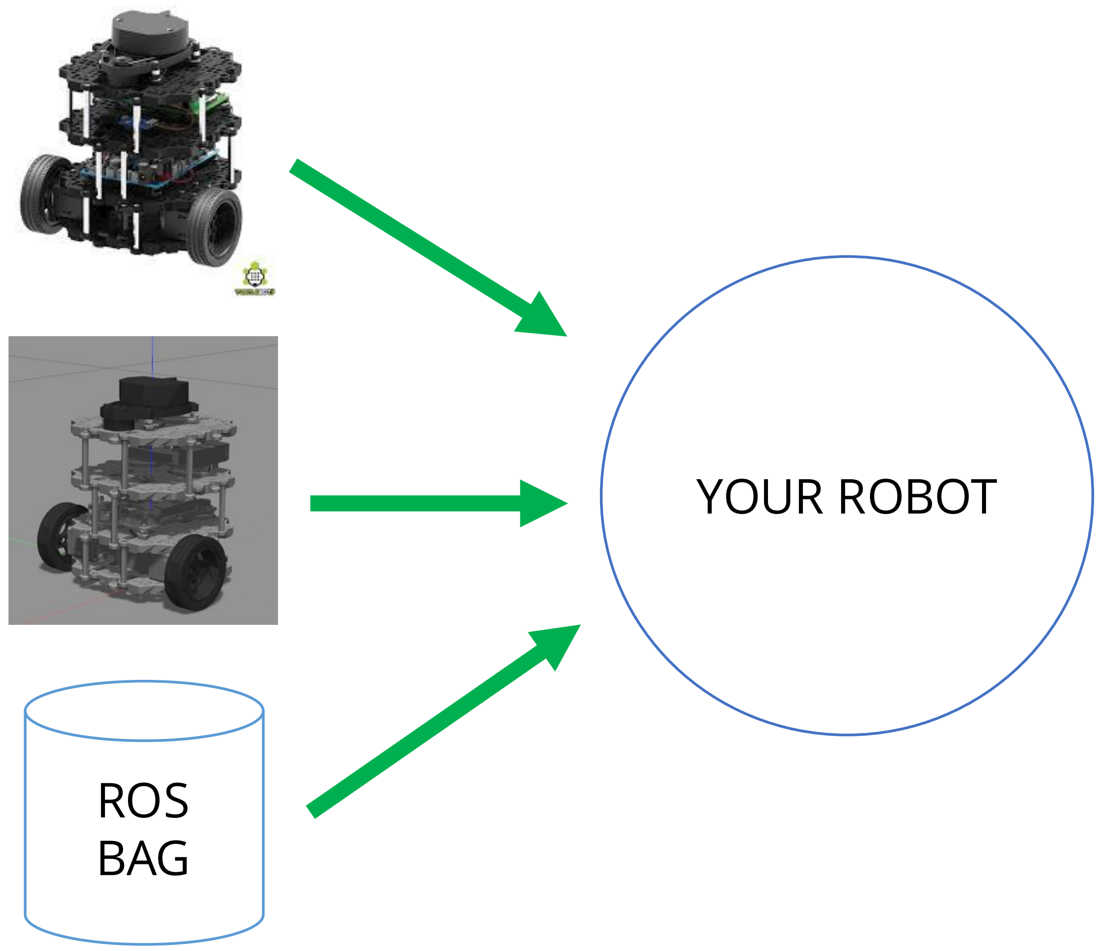
With the only constraint of using the same topics and the same msg format, you can switch between:

- real robots
- simulations
- pre-recorded data streams

without changing the other part of the robot code.

The rest of the robot code structure and of the nodes used remain the same.

Wrapping up

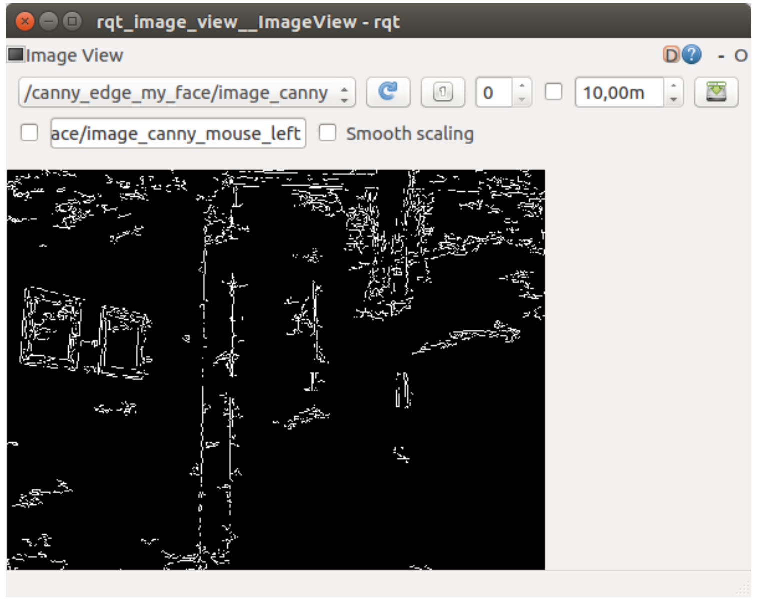


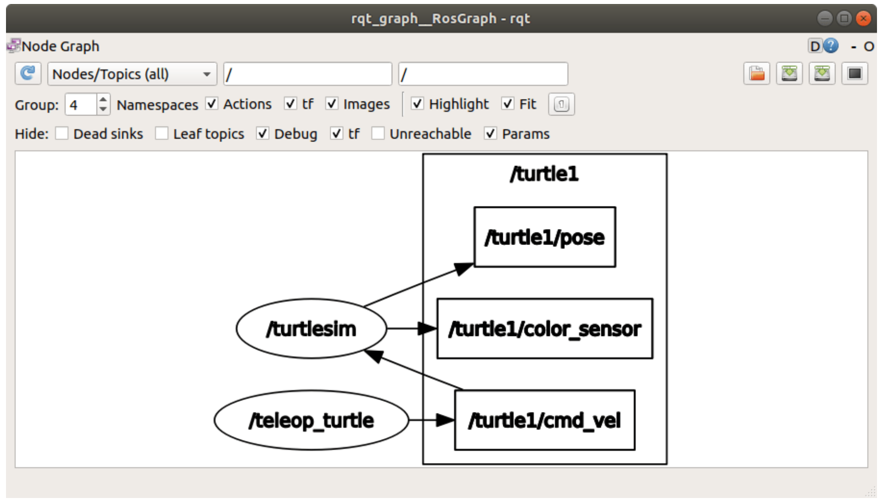
This has multiple (positive) side effects:

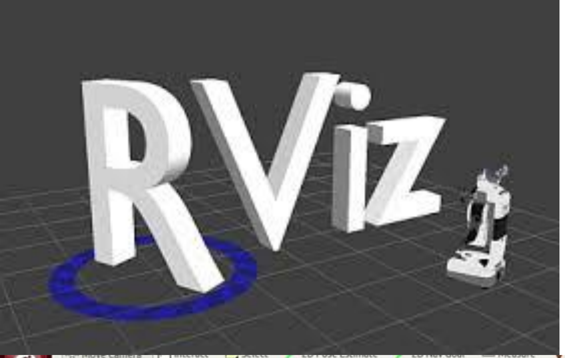
- You can focus on the specific task you want to develop
- You can develop a robot even without having a robot – use simulations
- You are not even forced to acquire no runs – just use datasets/rosbags

Debugging tool with ROS

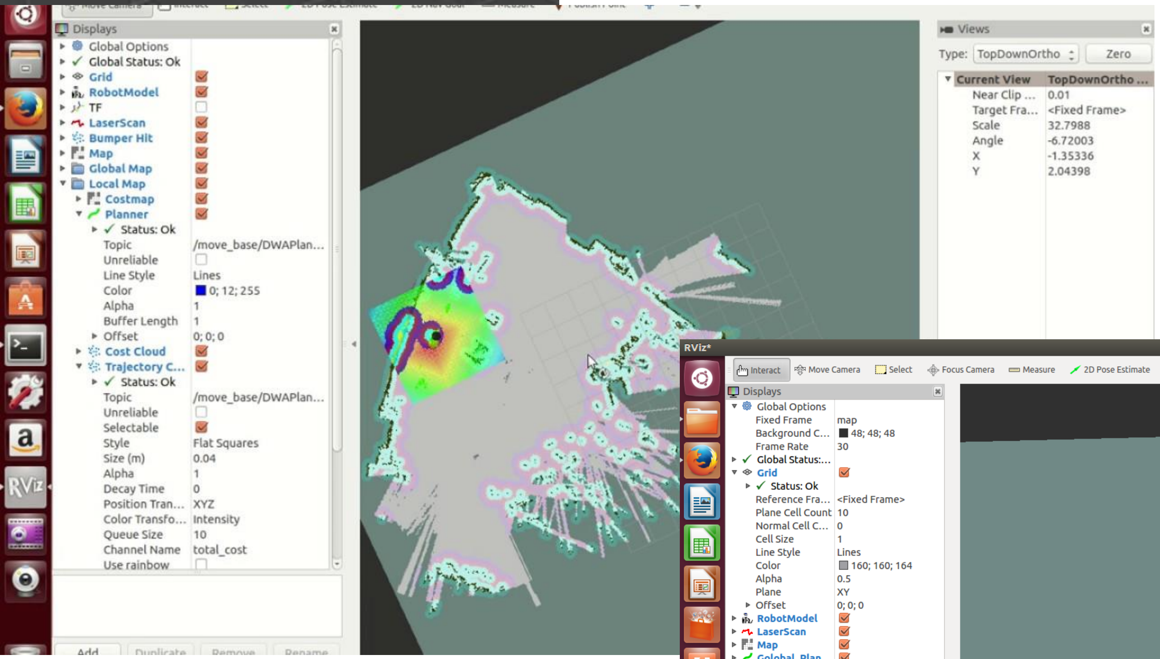
Even using simulations or ROS bags, robots are still complex. ROS offers several visualization tools that are useful for debugging of a complex system.


rqt_image_view__ImageView - rqt
Image View
/canny_edge_my_face/image_canny
0
10,00m
ace/image_canny_mouse_left
Smooth scaling





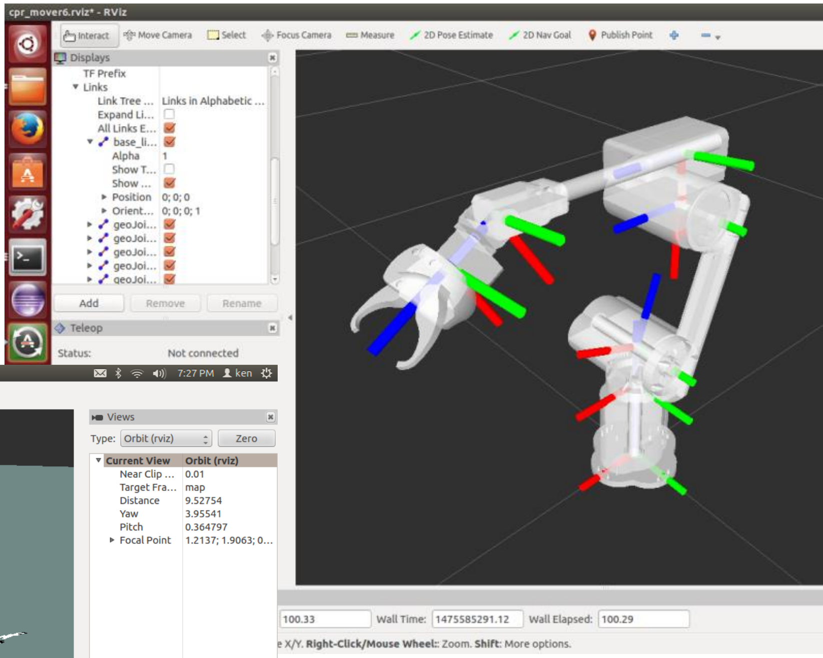
RViz is the main visualization tool of ROS.
It is used to display sensor readings, maps, cost maps, joints, TF, and, in general, to have an overview of the internal status of the robot and of all its sensors and nodes.



Displays
- Global Options
- Global Status: Ok
- Grid ✓
- RobotModel ✓
- TF
- LaserScan ✓
- Bumper Hit ✓
- Map ✓
- Global Map ✓
- Local Map ✓
  - Costmap ✓
  - Planner ✓
    - Status: Ok
    - Topic /move_base/DWAPlan...
    - Unreliable
    - Line Style Lines
    - Color 0; 12; 255
    - Alpha 1
    - Buffer Length 1
    - Offset 0; 0; 0
  - Cost Cloud ✓
  - Trajectory C... ✓
    - Status: Ok
    - Topic /move_base/DWAPlan...
    - Unreliable
    - Selectable ✓
    - Style Flat Squares
    - Size (m) 0.04
    - Alpha 1
    - Decay Time 0
    - Position Tran... XYZ
    - Color Transfo... Intensity
    - Queue Size 10
    - Channel Name total_cost
    - Use rainbow

Views
Type: TopDownOrtho Zero

Current View	TopDownOrtho ...
Near Clip ...	0.01
Target Fra...	<Fixed Frame>
Scale	32.7988
Angle	-6.72003
X	-1.35336
Y	2.04398

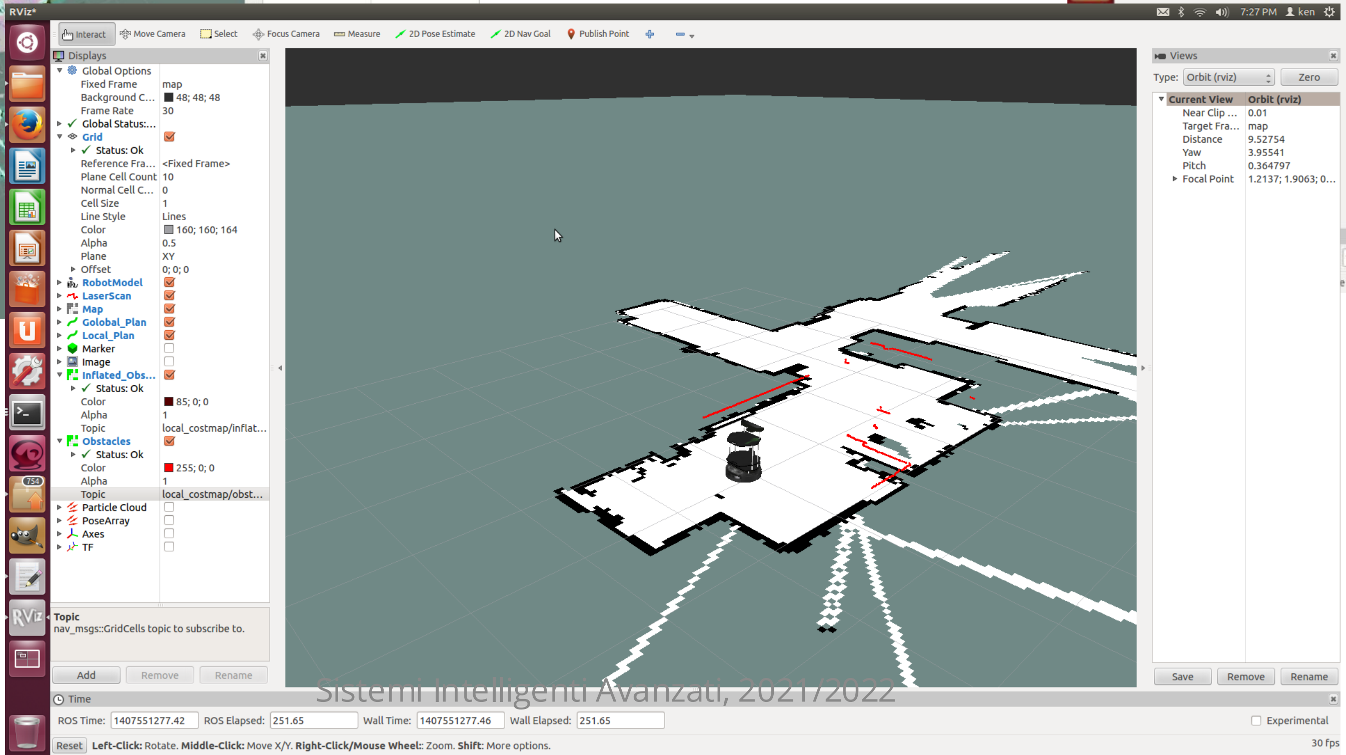


cpr_mover6.rviz* - RViz

Displays
- TF Prefix
- Links
  - Link Tree ... Links in Alphabetic ...
  - Expand Li...
  - All Links E... ✓
  - base_li... ✓
    - Alpha 1
    - Show T...
    - Show ... ✓
    - Position 0; 0; 0
    - Orient... 0; 0; 0; 1
  - geoJoi... ✓
  - geoJoi... ✓
  - geoJoi... ✓
  - geoJoi... ✓
  - geoJoi... ✓

Teleop
Status: Not connected

100.33 Wall Time: 1475585291.12 Wall Elapsed: 100.29



RViz*

Displays
- Global Options
  - Fixed Frame map
  - Background C... 48; 48; 48
  - Frame Rate 30
- Global Status:...
- Grid ✓
  - Status: Ok
  - Reference Fra... <Fixed Frame>
  - Plane Cell Count 10
  - Normal Cell C... 0
  - Cell Size 1
  - Line Style Lines
  - Color 160; 160; 164
  - Alpha 0.5
  - Plane XY
  - Offset 0; 0; 0
- RobotModel ✓
- LaserScan ✓
- Map ✓
- Golobal_Plan ✓
- Local_Plan ✓
- Marker
- Image
- Inflated_Obs... ✓
  - Status: Ok
  - Color 85; 0; 0
  - Alpha 1
  - Topic local_costmap/inflat...
- Obstacles ✓
  - Status: Ok
  - Color 255; 0; 0
  - Alpha 1
  - Topic local_costmap/obst...
- Particle Cloud
- PoseArray
- Axes
- TF

Topic
nav_msgs::GridCells topic to subscribe to.

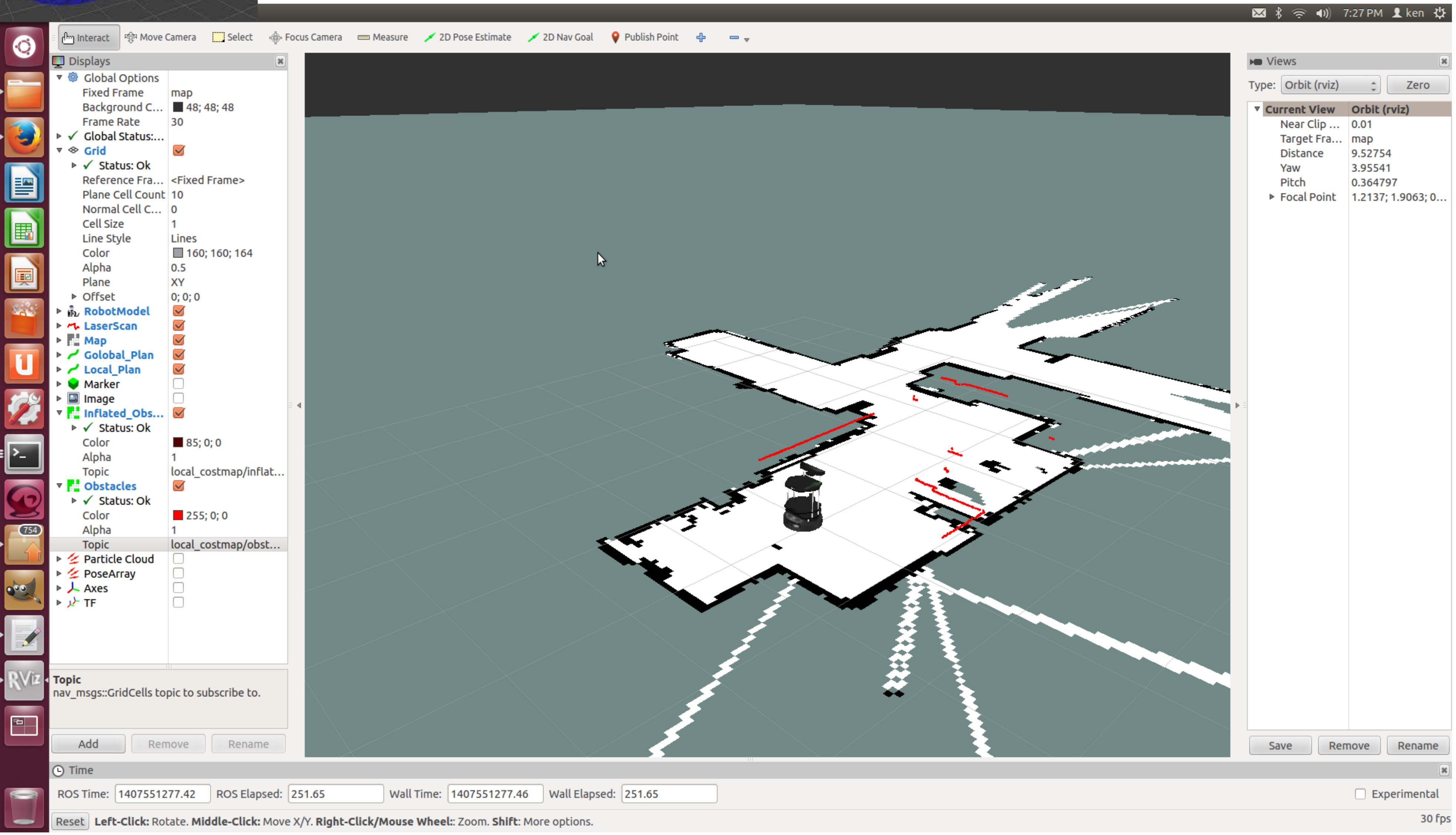
Views
Type: Orbit (rviz) Zero

Current View	Orbit (rviz)
Near Clip ...	0.01
Target Fra...	map
Distance	9.52754
Yaw	3.95541
Pitch	0.364797
Focal Point	1.2137; 1.9063; 0....

Time
ROS Time: 1407551277.42 ROS Elapsed: 251.65 Wall Time: 1407551277.46 Wall Elapsed: 251.65

Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options. 30 fps

ROS



Global Options	
Fixed Frame	map
Background C...	48; 48; 48
Frame Rate	30
✓ Global Status:...	
Grid	✓
✓ Status: Ok	
Reference Fra...	<Fixed Frame>
Plane Cell Count	10
Normal Cell C...	0
Cell Size	1
Line Style	Lines
Color	160; 160; 164
Alpha	0.5
Plane	XY
Offset	0; 0; 0
RobotModel	✓
LaserScan	✓
Map	✓
Golobal_Plan	✓
Local_Plan	✓
Marker	
Image	
Inflated_Obs...	✓
✓ Status: Ok	
Color	85; 0; 0
Alpha	1
Topic	local_costmap/inflat...
Obstacles	✓
✓ Status: Ok	
Color	255; 0; 0
Alpha	1
Topic	local_costmap/obst...
Particle Cloud	
PoseArray	
Axes	
TF	

Topic
nav_msgs::GridCells topic to subscribe to.

Type: Orbit (rviz)

Current View	Orbit (rviz)
Near Clip ...	0.01
Target Fra...	map
Distance	9.52754
Yaw	3.95541
Pitch	0.364797
Focal Point	1.2137; 1.9063; 0...

ROS Time: 1407551277.42 ROS Elapsed: 251.65 Wall Time: 1407551277.46 Wall Elapsed: 251.65 Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options. 30 fps

Rviz is “de-facto” a debugging tool for ROS.

- visualize sensor readings
- visualize maps
- check topic naming
- send robot to location
- see robot status (localization)
- check TF
- …

Autonomous navigation with ROS

Robot Navigation

Navigation is the ability to determine the position of the robot and to plan a path towards a goal location.

Navigation is both a core capability of autonomous mobile robots and a complex one.

ROS

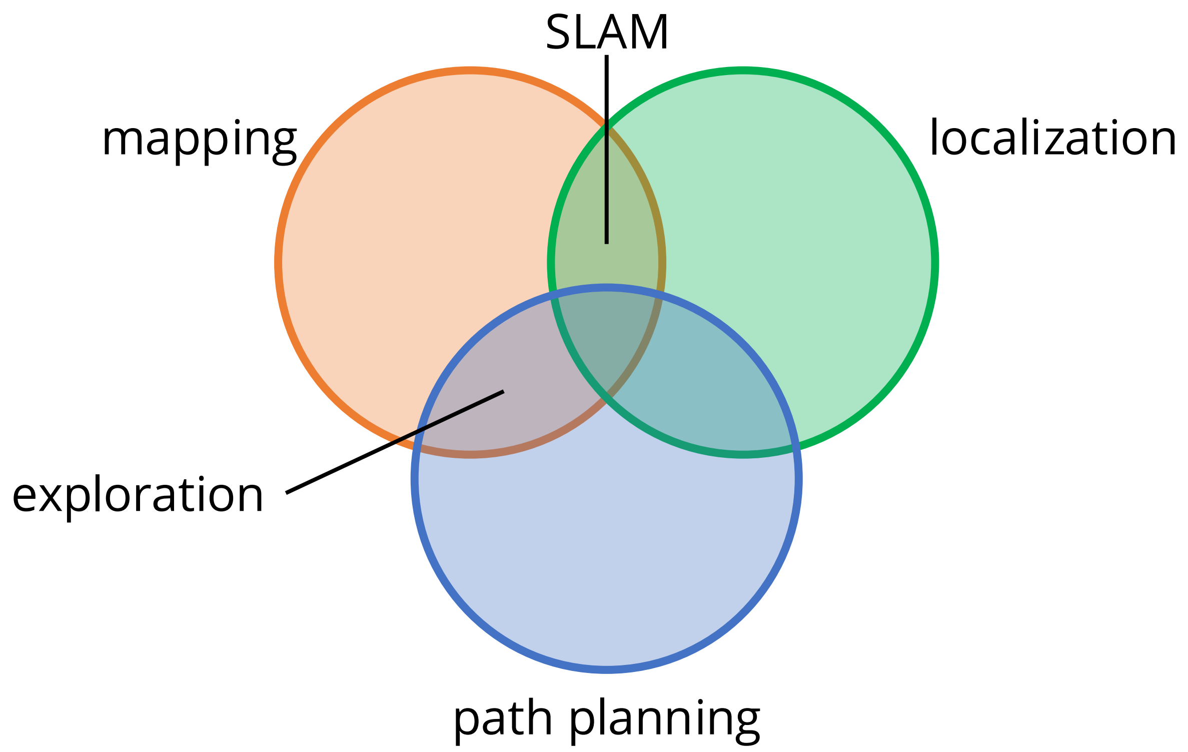
ROS and navigation

- how navigation is done is based on sensors mounted on the robot
- navigation sub-tasks depend on which is the goal of the robot (USAR, service robot, teaching, ...)
  (e.g. a robot working in close contact with humans, should take that into account)

ROS provides support in both directions:

- integration and drivers for sensors
- ready to use modules

From sensors to navigation



From Stachniss, Robot Exploration and Mapping

for moving autonomously the robot should be able to understand the environment from its sensor measurement

SLAM: Simoultaneous Localization and Mapping

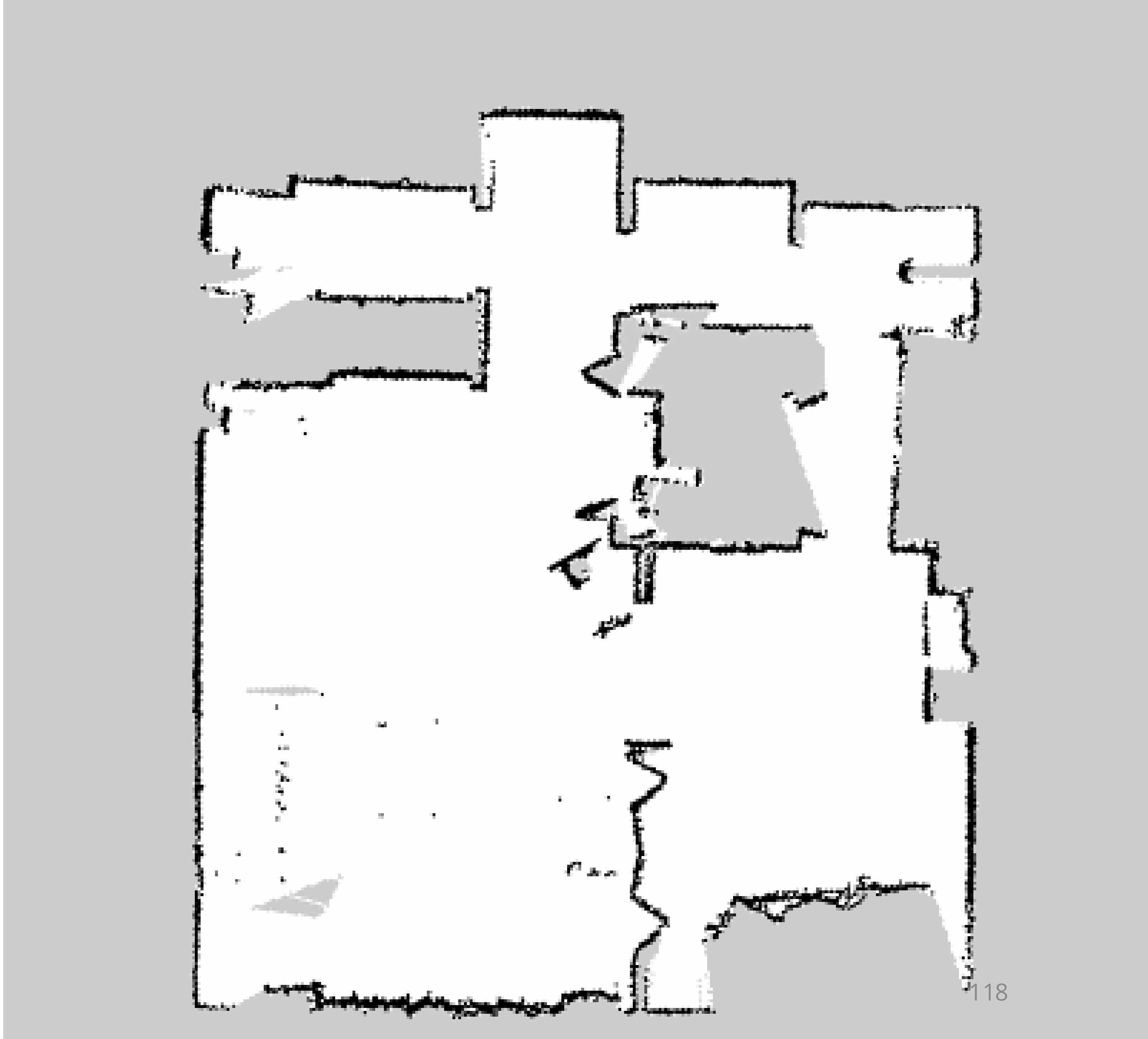
SLAM approaches can be different w.r.t the type of the robot:

- indoor
- outdoor
- marine (water-surface or submarine)
- underground
- …

The type of map built (2D/3D) and the type of sensors used for mapping (2D/3D lidars, vision, sonars, …)

2D SLAM

- used for indoor environments
- 2D grid maps
- robust
- available and ready-to-use solutions
- 2D lidar as source (cheap and reliable)
- most algorithms (e.g. planning) work with such representation



2D SLAM in ROS

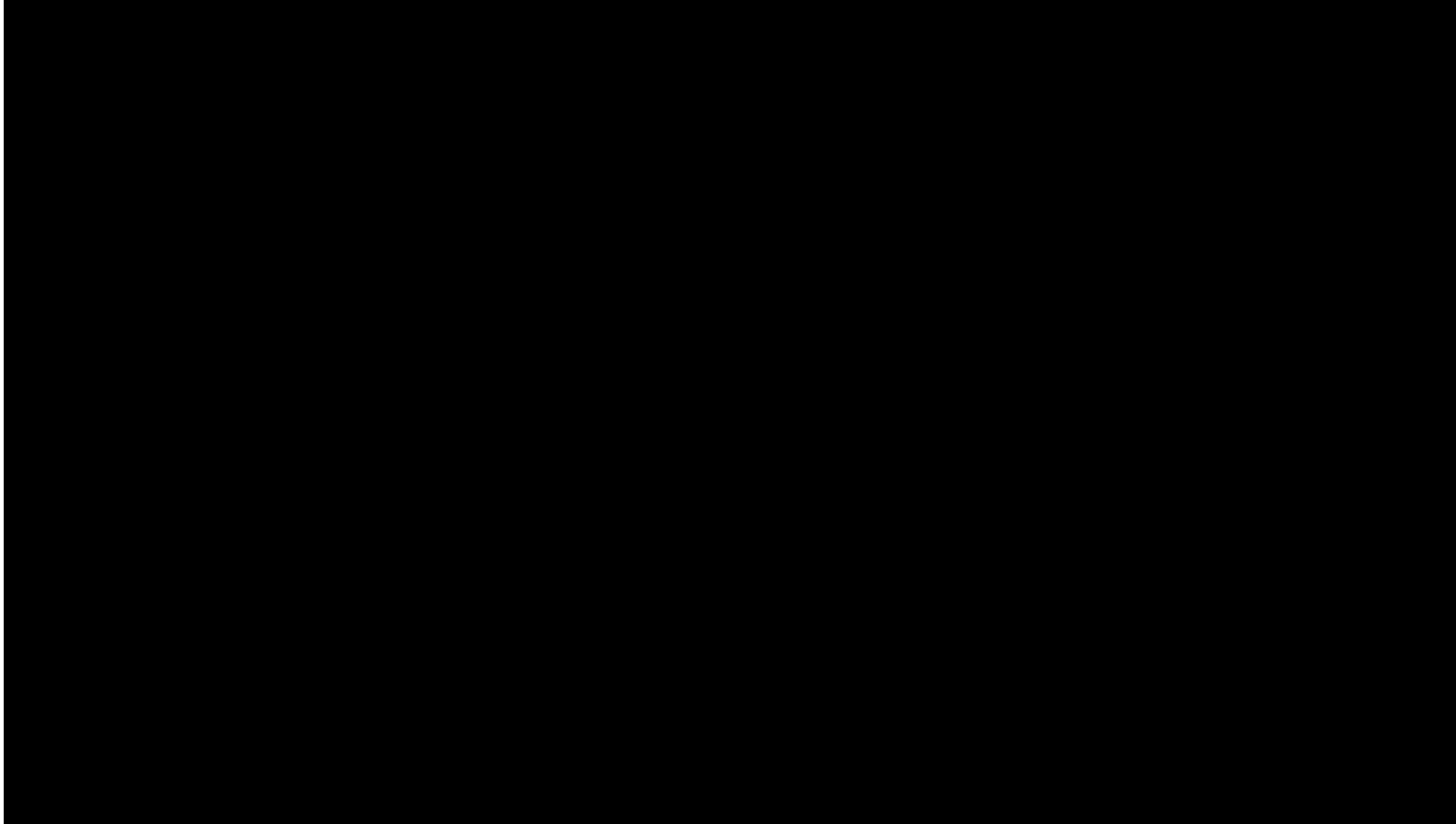
- several available methods – widely used, tested and robust
- need parameter configuration, but it is not that hard + docs
- Gmapping [Grisetti et al, T-RO, 2009], Hector SLAM, Cartographer, R-TabMap
- works reasonably well with a lot of different robot platform/settings, are robust to changes and clutter (noise in sensors and furniture, …), complex and large-scale environments, …

ROS

2D SLAM assumptions

- no-knowledge of the environment
- the map is built incrementally
- save and use the map later for future uses (scalability issues)
- the environment is static (open/closed doors)
- dynamic changes (e.g. people) can be filtered out (while mapping) and are not present when a static map is used…
- …low-freq dynamic changing (e.g. doors) and static changing could jeopardize robot localization and navigation

Example: Cartographer SLAM



RTAB-Map [Labbè – Michaud, JFR, 2019]

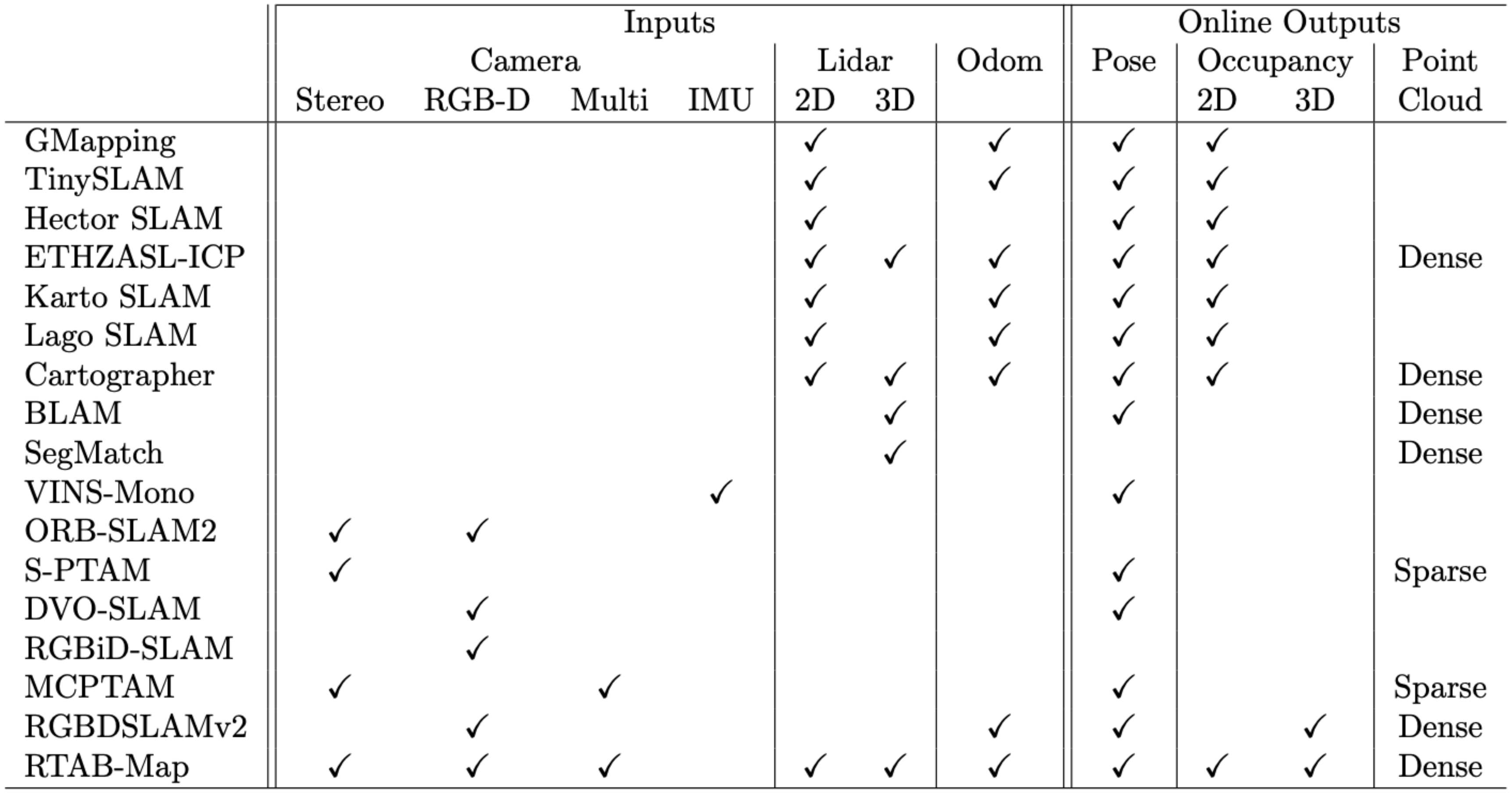
Offers both 2D and 3D SLAM

+

2D (lidar) and 3D (VIO) odometry

+

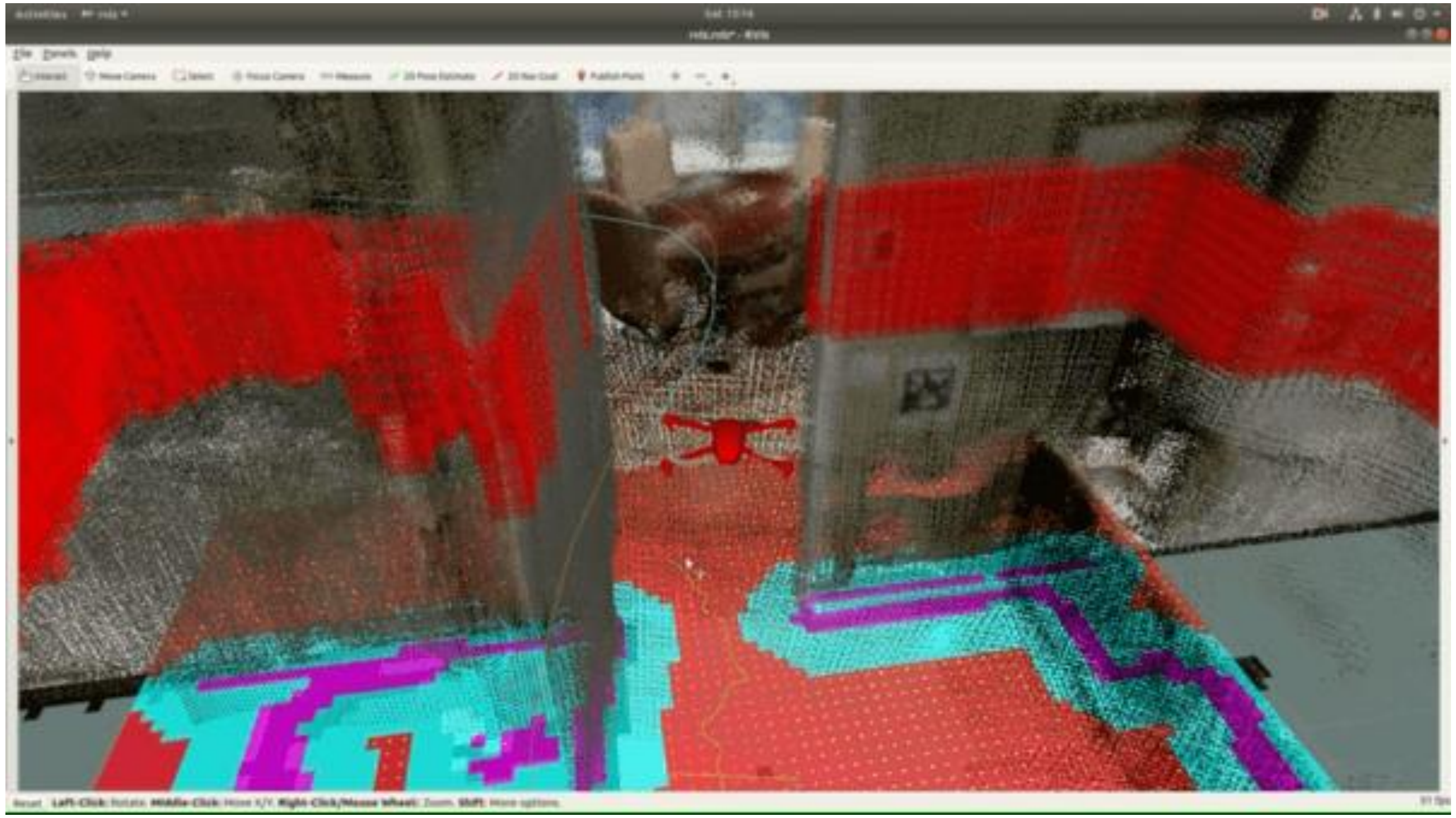
different representations



	Inputs							Online Outputs			
	Camera				Lidar		Odom	Pose	Occupancy		Point
	Stereo	RGB-D	Multi	IMU	2D	3D			2D	3D	Cloud
GMapping					✓		✓	✓	✓		
TinySLAM					✓		✓	✓	✓		
Hector SLAM					✓			✓	✓		
ETHZASL-ICP					✓	✓	✓	✓	✓		Dense
Karto SLAM					✓		✓	✓	✓		
Lago SLAM					✓		✓	✓	✓		
Cartographer					✓	✓	✓	✓	✓		Dense
BLAM						✓		✓			Dense
SegMatch						✓					Dense
VINS-Mono				✓				✓			
ORB-SLAM2	✓	✓									
S-PTAM	✓							✓			Sparse
DVO-SLAM		✓						✓			
RGBiD-SLAM		✓									
MCPTAM	✓		✓					✓			Sparse
RGBDSLAMv2		✓					✓	✓		✓	Dense
RTAB-Map	✓	✓	✓		✓	✓	✓	✓	✓	✓	Dense

RTAB-Map [Labbè – Michaud, JFR, 2019]

Offers both 2D and 3D SLAM
+
2D (lidar) and 3D (VIO) odometry
+
different representations



ROS Navigation stack

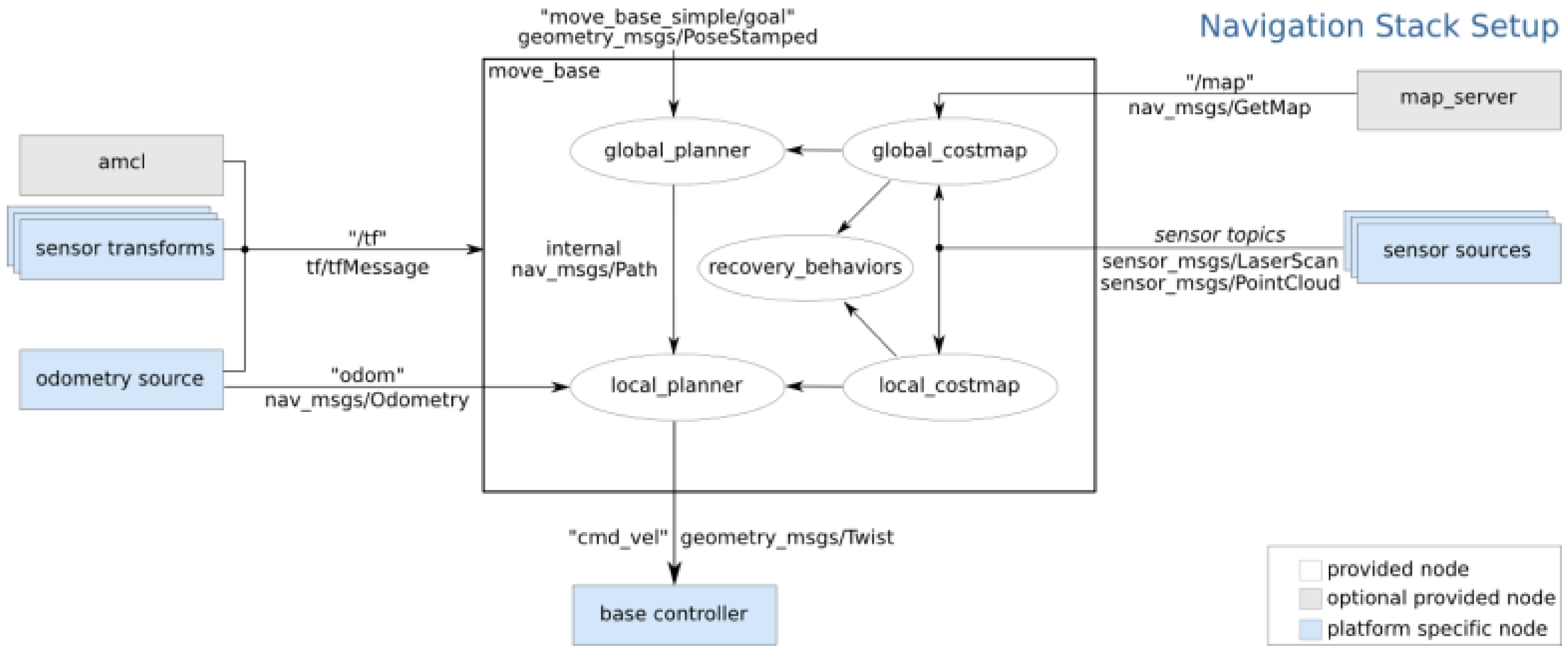
In your ROS configuration need:

- A robot
- Its sensors (proprioceptive/exceroceptive)
- Odometry
- A localization algorithm
- A map of the environment (or do SLAM)

The ROS navigation stack handles this setting and allows <u>path planning</u> and <u>path execution</u>; it implements a classic navigation framework.

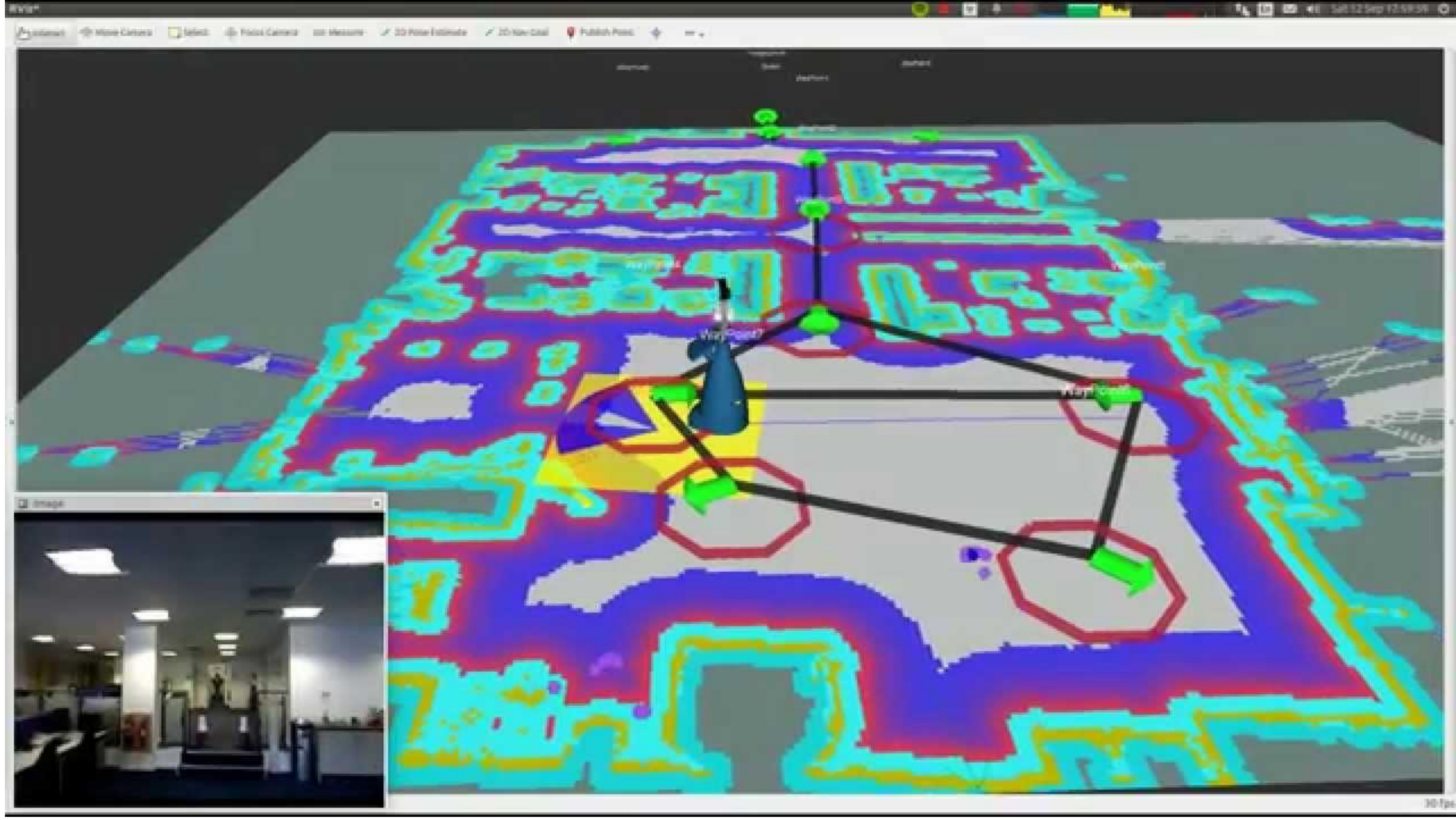
The core of the framework is called move_base

ROS Navigation Stack



Costmaps

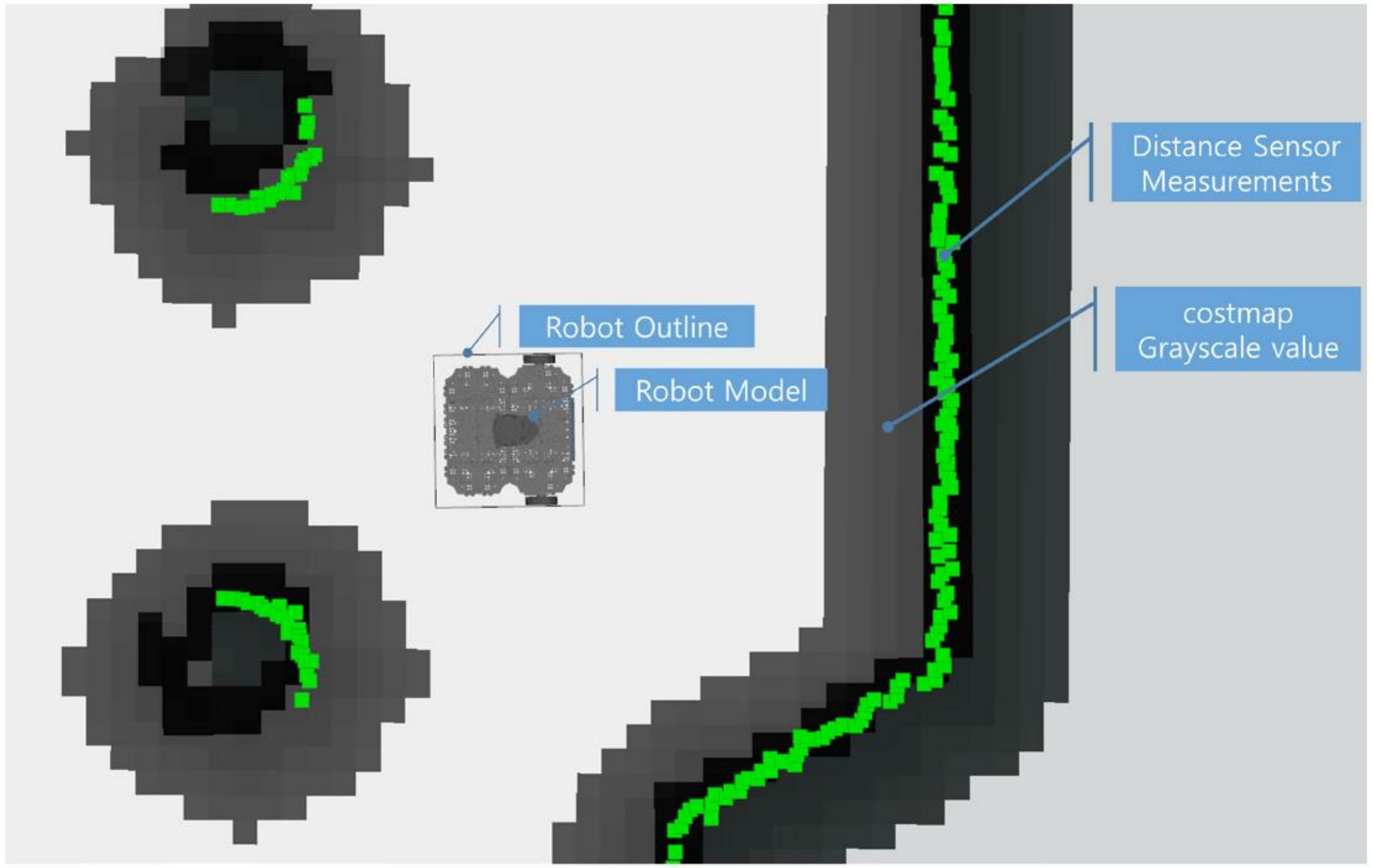
- the metric map is inflated according to the robot structure so the robot can perform a safe navigation



- in this way simpler paths in open areas are preferred to "costly" paths (close to obstacle or doors) where the robot may get stuck
- several methods to do so (e.g. robot footprint, inflate obstacles)

Costmaps

- the metric map is inflated according to the robot structure so the robot can perform a safe navigation



- in this way simpler paths in open areas are preferred to “costly” paths (close to obstacle or doors) where the robot may get stuck
- several methods to do so (e.g. robot footprint, inflate obstacles)

ROS

Deal with uncertainties and dynamics

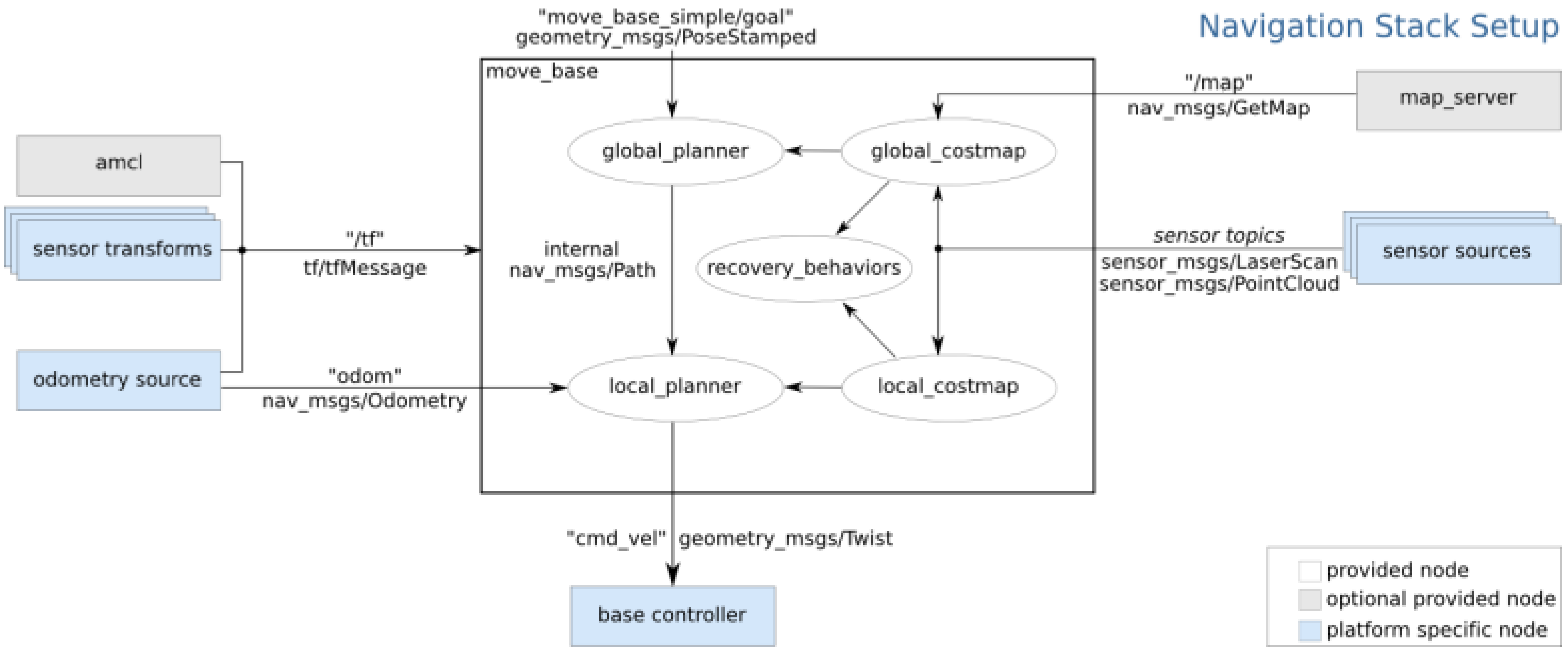
The robot plans its path in the static map but

- changes usually happens (doors open/close)
- new obstacles may appear (people, animals, children)
- the robot movements execution are very different w.r.t. the initial goal
- …

Solution: complement the (ideal) map with local information coming for sensors that address such issues

- obstacle avoidance
- local map refinement based on recent sensor readings

ROS Navigation Stack



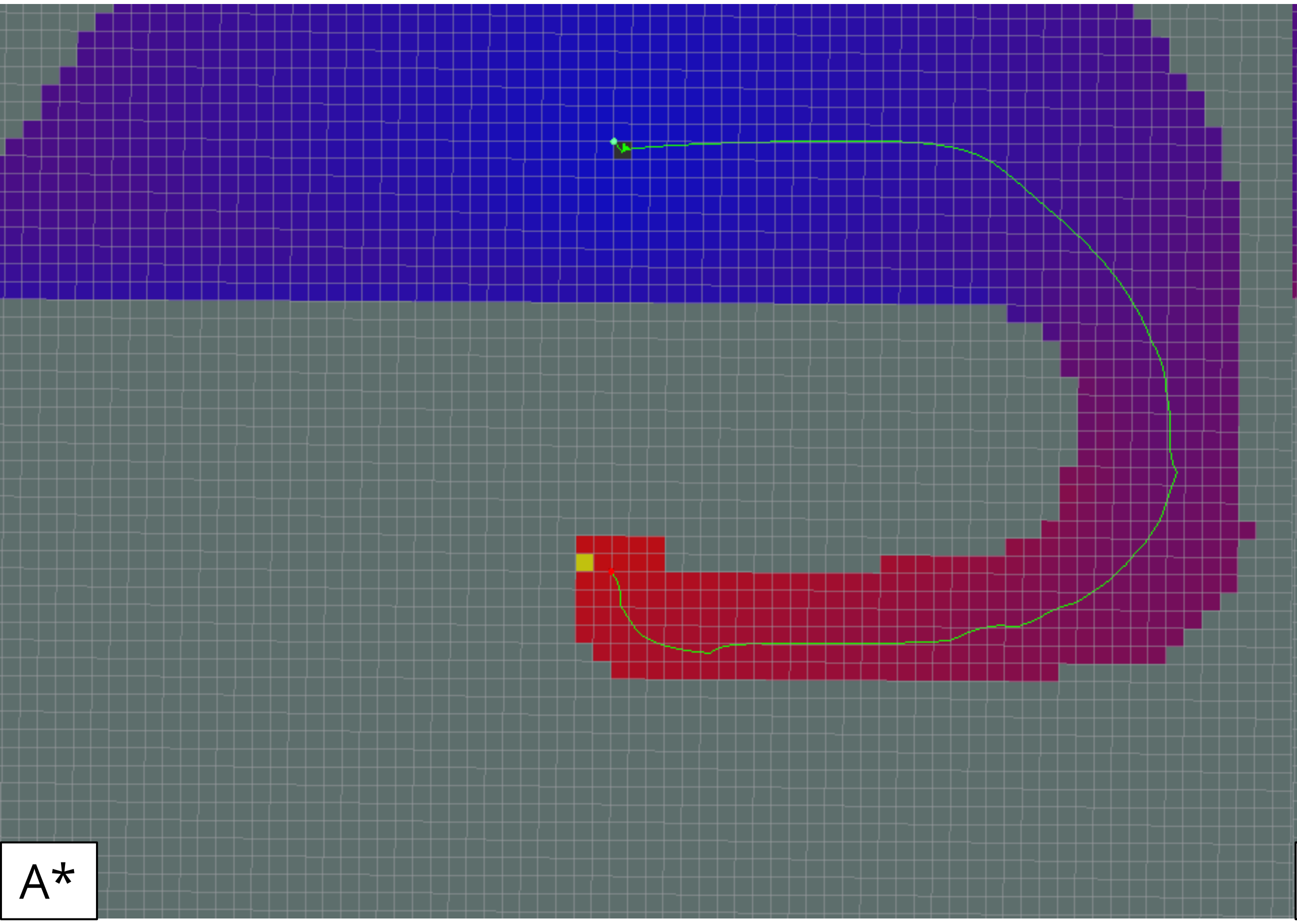
Global and local planner

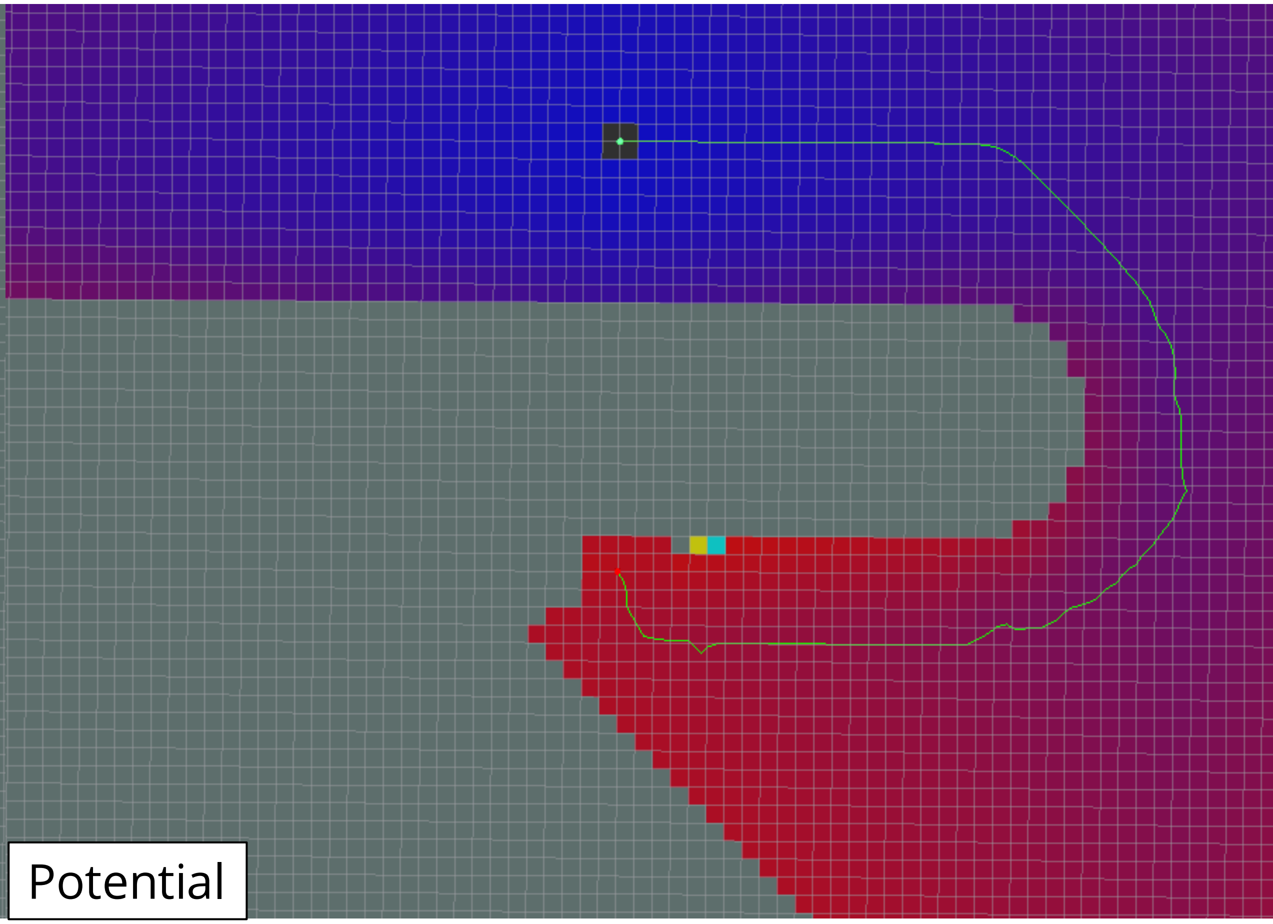
- global plan → identifies the long-term path that eventually will lead the robot to the goal
  works at low frequency, using 2D lidar data
- global costmap → used for path planning, based on the static metric map
- local plan → identify the next moves that the robot has to perform in order to follow the global path
  works at high frequency
- local costmap → centered on the robot, integrates all of the sensors of the robot (2D lidar, RGBD data, bumpers) that are needed to constantly adapt the local plan

Global planner and costmaps

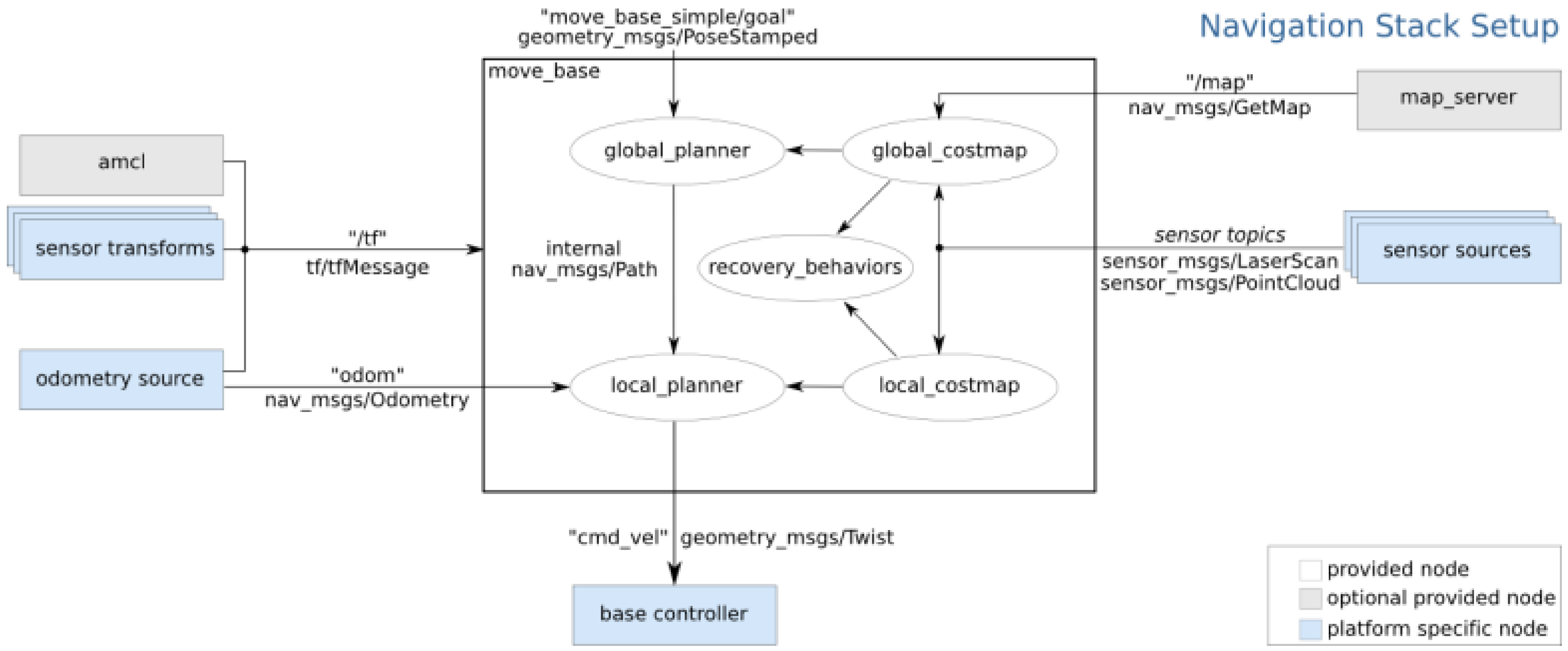
- implements several planning algorithms, use the one you want and that is most suited for your application
- costmaps also can be tuned in several ways according to your robot configuration
- you can visualize with RViz the path decided by the robot
- the global path could become outdated – replanning is also used

Global Planner





ROS Navigation Stack

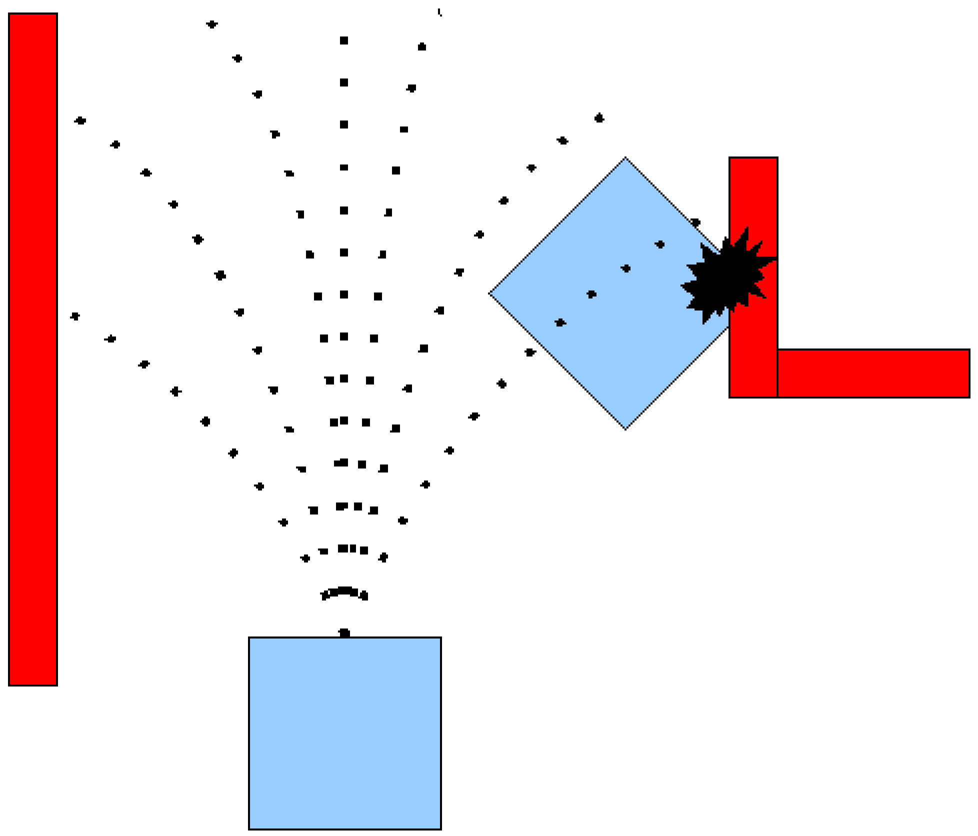


DWA local planner

The local planner package is a controller that drives a mobile base in the plane and connect the path planner to the robot.

Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location.

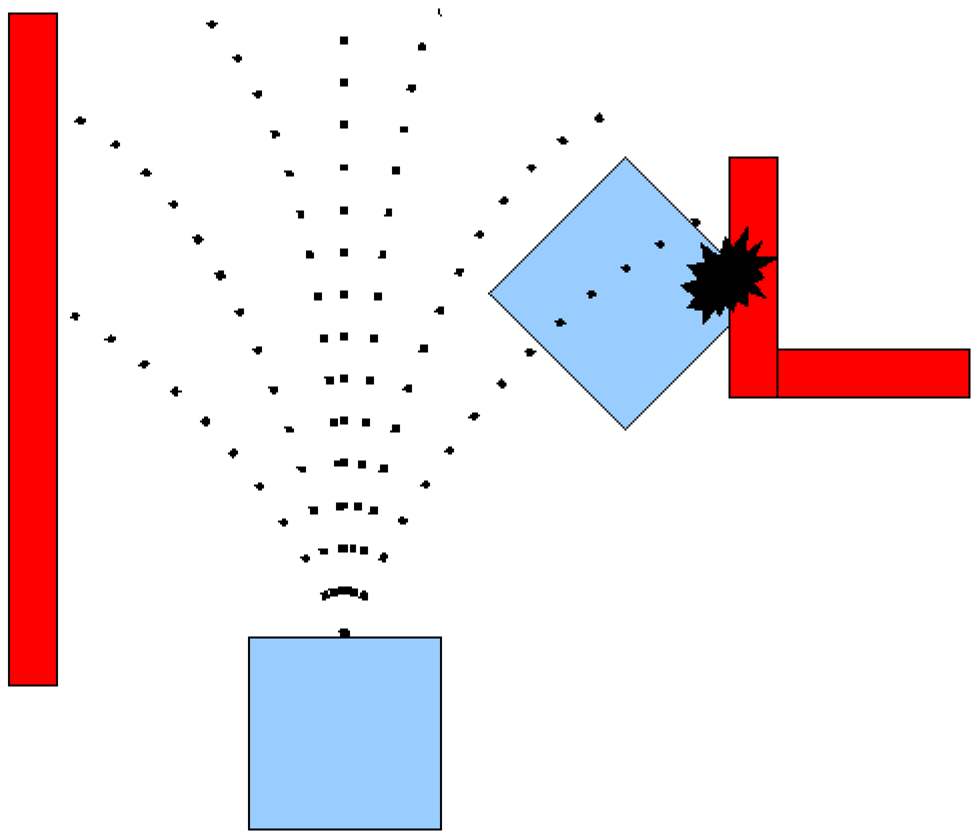
Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map.



DWA local planner

This value function encodes the costs of traversing through the grid cells.

The controller's job is to use this value function to determine *dx,dy,dtheta* velocities to send to the robot.

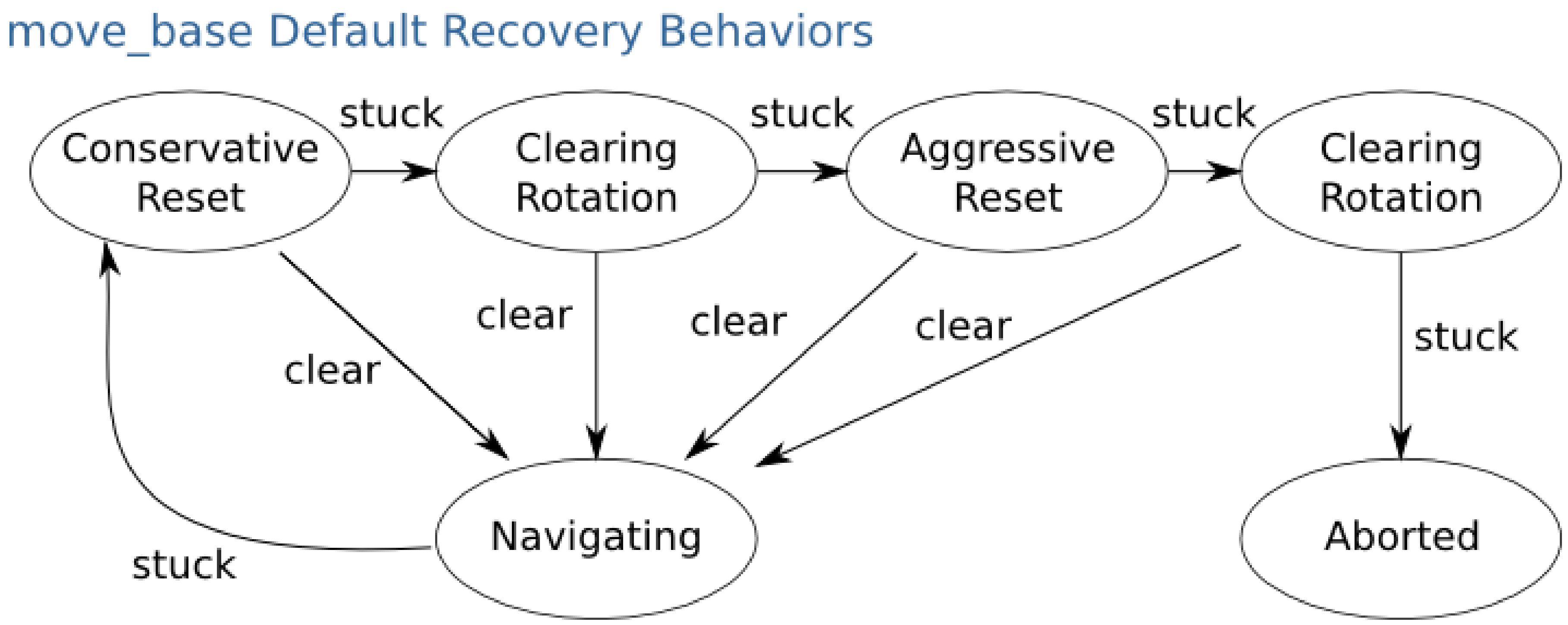


Handling failures

- despite the integration of local and global plan execution, the robot may get stuck → is not able to move and continue to execute its path
- this happens often in narrow passages (doorways), when a lot of rotations are involved, or with dynamic obstacles (people, is too close to an obstacle to safely move)
- the robot should be provided with mechanism to solve autonomously such issue and to restart following its path
- otherwise, a human intervention is needed

The navigation stack gives you a set of behavior for this

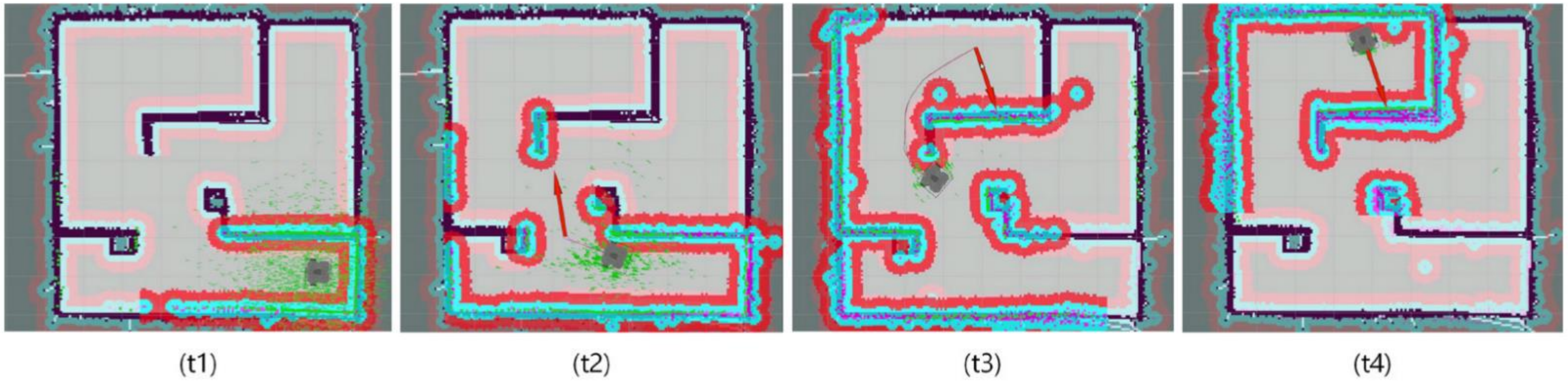
Recovery Behaviors



Recovery behaviors are executed when the robot is stuck or cannot proceed to the goal (cannot execute the path or cannot compute the path).
They try to free the robot from a dangerous position (e.g. too close to an obstacle) or to “clear” the costmaps (e.g. a noisy reading, a user was in front of the robot, an obstacle that was there and it is not there anymore)

ROS

An example of global and local costmaps + AMCL

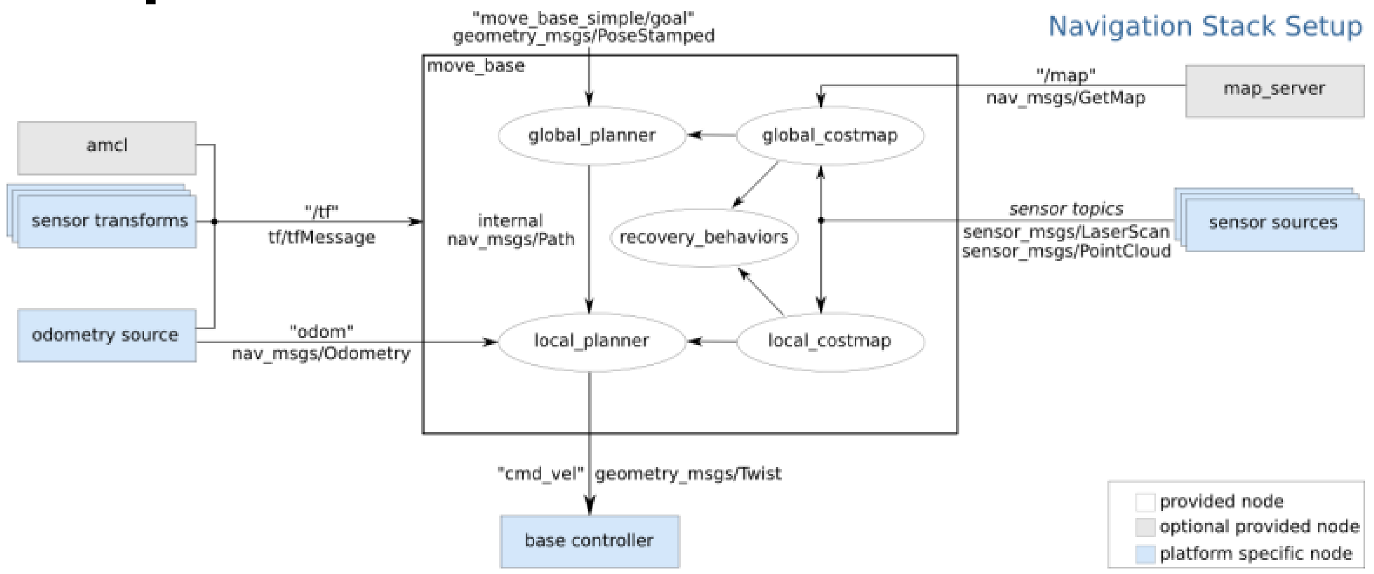


ROS Navigation wrap up

All the required modules for having a robot moving autonomously are available and ready-to use in ROS.

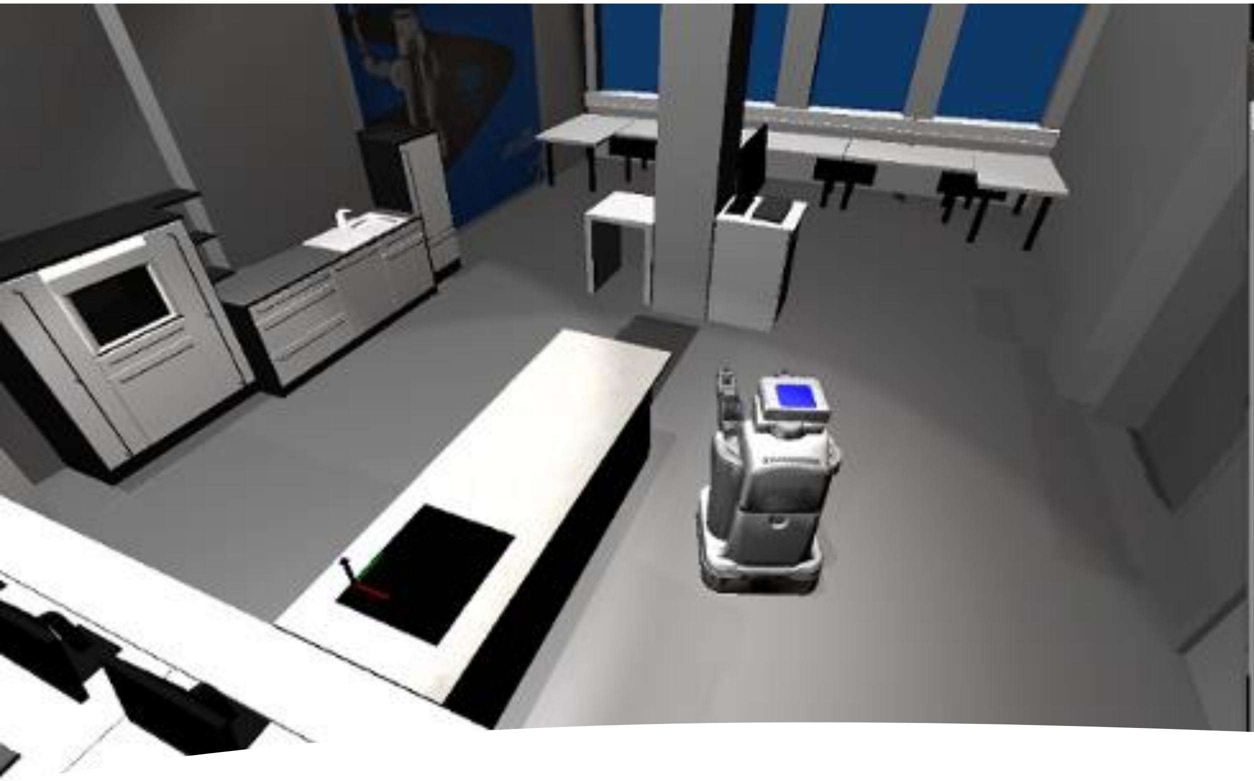
You just need to:

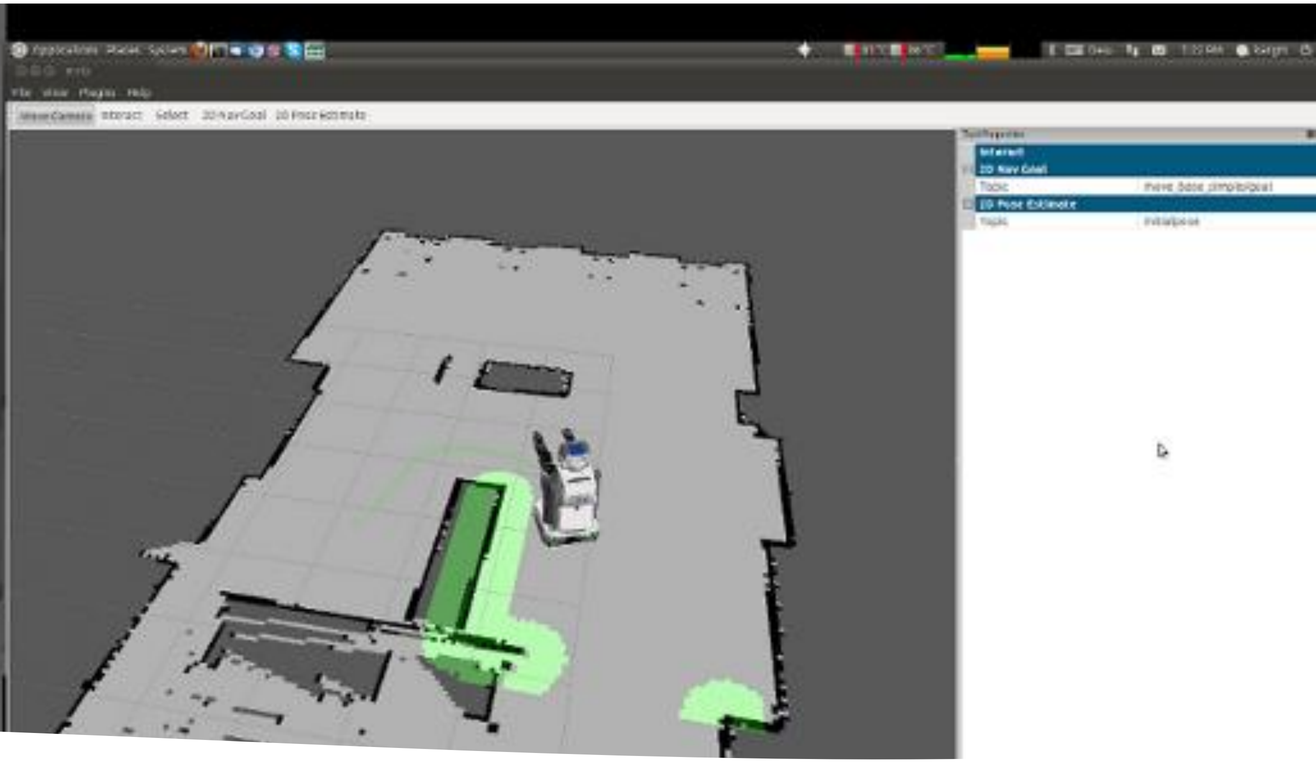
- select sensors and a robot
- pick up a SLAM algorithm and make a map
- use a localization mechanism
- use the navigation stack



ROS Navigation wrap up

You can use simulations and test everything you've seen by yourself in a couple of afternoons





RViz is very useful to test and understand what happens.

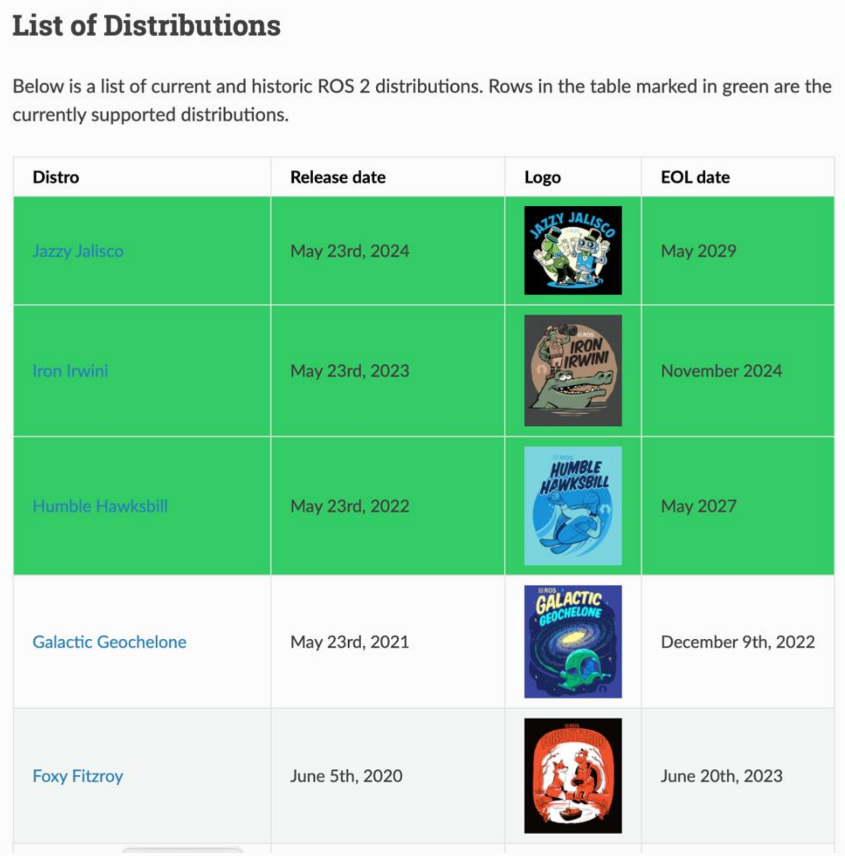
From ROS to ROS2

- Last ROS version released: Noetic in 2020 (**stable but old**)
- Now we are migrating to ROS2 (**more or less stable**)

New stuff:

- QoS
- Security
- Multi-Robot
- Multi-Thread
- Real-Time
- Behaviour Trees

Rule of thumb: choose ROS2 for a new project



List of Distributions

Below is a list of current and historic ROS 2 distributions. Rows in the table marked in green are the currently supported distributions.

Distro	Release date	Logo	EOL date
Jazzy Jalisco	May 23rd, 2024		May 2029
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		June 20th, 2023

Sources - References

- Wiki.ros.org
- ROS Robot Programming
  A Handbook Written by Turtlebot3 Developers
  (available at http://www.robotis.com/service/download.php?no=719)
- Robotis Turtlebot3 documentation
  http://emanual.robotis.com/docs/en/platform/turtlebot3/getting_started/
- Jason O'Kane, A Gentle Introduction to ROS
  https://cse.sc.edu/~jokane/agitr/agitr-letter.pdf

Sources - References on navigation

ROS wiki navigation stack home page, with a lot of tutorials and documentation

- http://wiki.ros.org/navigation
- http://wiki.ros.org/navigation#Tutorials
- http://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack